Using Concurrency and Parallelism Effectively – I

Jon Kerridge

Jon Kerridge

Using Concurrency and Parallelism Effectively – I

Using Concurrency and Parallelism Effectively – I
1st edition

ISBN 978-87-403-0802-0

Contents

Preface

The aim of this book is to show both students and practitioners that concurrent and parallel programming does not need to be as hard as it is often portrayed and in fact is often easier than building the equivalent sequential system. This will be achieved by presenting a set of example systems that demonstrate the underlying principles of parallel system design based upon real world examples. Each chapter will discuss the complete implementation of such a system, rather than presenting fragments of solutions. The approach will therefore be founded in principled engineering rather than a detailed exploration of the scientific underpinning. The science has been explored in many books but these have not demonstrated the engineering aspects of actually designing and building parallel systems.

For the purposes of this book; Concurrent means a system built from a set of processes that execute on a single processor. Parallel means that more than one processor is used to execute the processes and these communicate over some form of network. Within a parallel system it is likely that some of the processors will run some processes concurrently.

The book will use as its underpinning parallel environment a package called JCSP (Communicating Sequential Processes for Java) that is available under the LGPL software licence from the University of Kent, Canterbury UK (Welch, 2002) (Welch, 2013). This package implements the Communicating Sequential Process concepts developed by Professor Hoare some 30 years ago (Hoare, 1978) in a form that makes them easily accessible to the programmer. The book's emphasis is on the engineering of parallel systems using these well-defined concepts without delving into their detailed theoretical aspects. The JCSP package essentially hides Java's underlying thread model from the programmer in a manner that allows easy implementation of concurrent and parallel systems. It is immaterial whether a process is executed concurrently or in parallel, the process definition remains the same. The JCSP implementation is essentially a re-implementation of the occam programming language (Inmos Ltd, 1988) developed for the Inmos Transputer. (Wikipedia, 2013).

The underlying theory for JCSP is based upon Hoare's Communicating Sequential Processes (Hoare, 1985) (Hoare, 1978) and is still the subject of research and development. CSP allows the designer to reason about the behaviour of their system, provided they use some well-defined patterns. In this book we shall be creating designs that use these design patterns to ensure that our designs behave correctly.

Understanding the principles behind parallel processing is an increasingly important skill with the advent of multi-core processors. Much effort has been made by processor manufacturers to hide the underlying parallel design techniques by providing tools that will take an existing code and extract some parallelism from it. This hides the real need to actually design and build parallel systems from the outset. Far too many people have been put off concurrent and parallel programming because they believe that they have to understand the underlying thread model supplied as part of the language or operating system environment. The goal of the book is to dispel all these misconceptions and show that parallel systems can be built quite easily with a very few simple design patterns and that such parallel systems can be easily implemented on a single processor or a collection of networked processors. Furthermore the advent of multi-core processors means that we can now start to build genuinely parallel systems for the commonest desktop workstations in which we can exploit the inherent parallelism more. The extension to a network of multi-core processors becomes even easier. Equally important is that the same design principles can be used to build mobile systems that permit interactions between mobile devices and fixed services using wireless (wi-fi) and Bluetooth technology.

Background

The book results from a module taught during the spring semester to masters and undergraduate students, though the approach would be applicable to professional programmers. As part of the module, students were asked to complete a practical portfolio and these exercises are included in the book. The source coding for all the examples and exercises is also available.

Parallelism, as such, has been a topic that has been avoided by many educational establishments, possibly because of the lack of appropriate software environments and approachable tools. It also suffers from the underlying models being overly complex and difficult to reason about. This book addresses these issues by providing a model of parallel programming based in the Object Oriented paradigm that builds upon many years of research into the design and implementation of parallel systems. It attempts to demonstrate that parallel programming is not hard and perhaps should be considered as a more appropriate first design environment for the teaching of programming as it is closer to the way in which humans understand and investigate solutions to problems.

Why Java and Groovy and Eclipse?

Java is widely used and, increasingly in mobile applications, where the ability to build parallel systems is crucial. Most user interfaces are in fact concurrent in nature but you would not think so given the contortions a Java programmer has to go to make them appear sequential. Groovy is a scripting language for the Java platform (Kerridge, et al., 2005). The addition of a small number of helper classes has made the underlying parallel design concepts far more easily accessible as the amount of code that has to be written to create a working application is dramatically reduced. It is assumed that the reader of this book has some familiarity with Java and Java-like languages and sufficient understanding of object-oriented principles to cope with data encapsulation and the concept of get() and set() methods. An awareness of the java.awt package will be of benefit when user interfaces are discussed.

Eclipse is a commonly used Integrated Software Development Environments and as such has the appropriate plug-ins for Java and Groovy which have been extended to include the Groovy Parallel helper classes.

Example Presentation

The description contained in the text contains code snippets taken from the complete source of all the examples used in the book. In particular, the associated Eclipse project contains all the sources for the Examples and Exercises. Each example is presented as a listing that contains line numbers. These line numbers are the same as if the listing were opened in Eclipse. Package name and library import statements have been removed from the examples presented in the book, which is why most listings do not start at 1 but at the first line that contains pertinent content to explain the example. The notation {n} refers to line n in the cited Listing and {n-m} refers to lines n to m.

Organisation of the Book

The material is divided into two books. In this first book the basic concepts are presented and naturally fall into two parts. Part 1 introduces the basic concepts of the environment used in the book and comprises chapters 1 to 8. Part 2 demonstrates how these concepts are used to build reliable systems that run on a single processor and comprises Chapters 9 to 14.

The second book, available from the same web site, comprises the rest of the material and explains how systems can be constructed that run on multiple processing nodes. Chapter 15 provides the basic introduction to multiprocessor systems by taking some of the examples developed in this book and running them on a multi-processor system. Thereafter more complex examples are created that solve a variety of parallel programming problems.

Supporting Materials

The necessary libraries for the packages used are available from the same web site as the book's text. This comprises jar files for the JCSP and Groovy Parallel packages. Documentation for these packages is also provided. A readme text file that describes the library folder structure required is also provided so readers can create the required learning environment quickly and easily.

The source of all the examples is provided in the form of an archived Eclipse project, which can be imported directly into Eclipse, once the required Eclipse environment has been constructed as described in the readme file.

The source code comprises two projects; one called ChapterExamples that contains the source of all the examples used throughout the text. The other, ChapterExercises, contains supporting material for the Exercises given at the end of some of the Chapters. In these classes there are comments placed where the reader is asked to supply the required coding.

The source of the supporting material is also available in a zip file for non-Eclipse users.

Acknowledgements

This book started as a result of discussions I had with my colleagues Ken Barclay and John Savage (Barclay & Savage, 2006). It was they who introduced me to the delights of Groovy and helped me with the design of the Groovy Helper classes that are the basis of the programming structures used throughout the book. Kevin Chalmers, was originally my doctoral student who has now become a much valued colleague. We spent many happy hours discussing the net2 architecture that was the topic of his doctoral thesis (Chalmers, 2009). Kevin also undertook the review of the complete book, for which I am very grateful.

The group at the University of Kent, primarily, Peter Welch and Fred Barnes, has been the source of much stimulation and excellent ideas. It is their library, JCSP, which provides the language support that underpins this book. Another source of inspiration is the series of Conferences entitled Communicating Process Architectures that has been active since the mid-1980s and for which a large and extensive database of papers exists (http://wotug.org/).

Finally, to the students both undergraduate and postgraduate who have, by commenting on earlier versions of the content helped me to achieve the current state. Without the motivation they provide and the excitement they engender as they discover that parallel programming is not hard and in fact in many ways is easier than sequential programming. Also the many project students who have taken the earlier versions of the text, taught themselves the principles and then used them in a variety of interesting ways to solve challenging problems.

1 A Challenge – Thinking Parallel

This chapter introduces the subject by

- defining concurrency and parallelism
- discussing a motivating example based on a multi-player on-line game
- introducing the main concepts of processes, channels, alternatives and timers

Designing and building concurrent and parallel systems is easy, provided the appropriate language structures and methods are used. In fact it can be argued that designing concurrent systems is easier than designing sequential systems because they more closely follow the initial thought processes when a problem is posed. However, there is a widely held opinion in the wider computing community that concurrent programming is hard and should be avoided at all costs. This is undoubtedly true if you use any of the existing thread models to build concurrent systems. That is like using assembler to create a complex user interface accessing a database system. We need a higher level abstraction that allows the programmer to think concurrently and avoid the nitty-gritty of thread based programming. It is very hard to convert ideas into artefacts if you do not have the appropriate language capability. For too long programmers have been trying to design and build concurrent and parallel systems with inappropriate and dangerous language structures. The aim of this book is to demonstrate that a few simple language structures and one design pattern are all that is required to build parallel systems that are correct and about which we can reason. The language structures and design pattern are supported by a wealth of formal verification that allows the engineer to have confidence in their designs, without having to fully understand all the formal underpinning.

1.1 Concurrency and Parallelism

Concurrent means a system built from a set of components that execute on a single processor. The components interact with each other in some managed and controlled way to ensure that these interactions do not have unwanted side effects. These components or processes are said to be interleaved on the processor because only one process can execute at one time and the available processing resource is shared among the processes. At any one time more than one of the processes could be executed and the system designer should not make any assumptions as to which process will be allocated the processor. Parallel means that more than one processor is used to execute the processes and these communicate over some form of communication infrastructure. This could be a multi-core processor using memory or a distributed system running a TCP/IP network Within a parallel system it is likely that some of the processors will run some processes concurrently.

1.2 Why Parallel?

There are two reasons for wanting to think parallel. The obvious one enables the programmer to take advantage of modern multi-core processors and networks of workstations. This is generally aimed at obtaining better performance in solving a problem.

The less obvious reason is that, surprisingly, in many cases, it is easier to design a concurrent solution to a problem. The design process seems more natural as it involves just processing elements and the flow of data between these processing elements. The design process is essentially one of creating data flow diagrams. We shall discover that there is just one fundamental parallel design pattern used in the creation of these data flow diagrams or process network diagrams. A further advantage of this design process is that all the processing of data is contained within a processing element and thus the behaviour of it can be observed simply by looking at the definition of the processing element itself. Once the behaviour of a processing element is known it then becomes much easier to connect many of these together with a compositional style of network construction because data processing only takes place within a processing element. This then has the concomitant benefit on the design of data structures, because they become much simpler. They can be defined in a single class, with the required access and manipulation methods and treated as an abstract data type. There is no need to construct data objects that are 'thread safe' because any instance of a data object can only be accessed by a single processing element at any one time.

A further advantage of the approach is that the definition of the processing elements is independent of whether it will be executed concurrently, in a single core of a multi-core processor or in a distributed system. The way in which a processing element is invoked will change but not the definition.

This book focusses on the design of concurrent and parallel systems rather than the performance of the resulting solutions. Without good design and the ability to think parallel it is difficult to exploit the performance advantage that multiple processing elements give.

1.3 A Multi-player Game Scenario

The Pairs Game is a simple game played by families to improve children's spatial awareness and memory. A pack of cards is spread out face down in a regular pattern. A player turns over one card, and then a second. If the cards match in number and colour then the player retains that pair of cards. If the cards do not match then the player replaces the cards face down and another player can choose a pair of cards. Obviously as more cards are turned over the players 'learn' where pairs are located so they can increase the number of pairs they retain. The winner is the player with the most pairs.

For this implementation we shall assume that each player has their own workstation, connected to a network, through which they can interact with the game. In addition there is another workstation that has the ability to create new games and also to manage the interactions between the players. One aspect of the children's version is that it also teaches them to take turns and the patience to wait without giving away the location of a pair. An initial representation of the game architecture is given in Figure 1-1, with three players.

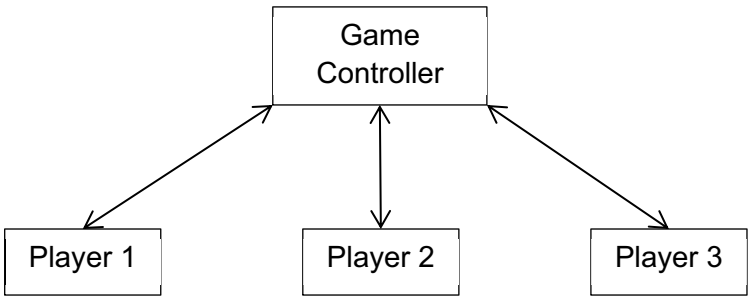


Figure 1-1 Initial Game Architecture

We assume that messages, or data, are going to be passed between each of the Players and the Game Controller but there is no direct interaction between the players. The design process then becomes one of specifying the data and messages that are passed between the processing elements, after which they can be specified and implemented. Each processing element can be tested in isolation once we known the data and messages it is to receive, send and manipulate.

1.3.1 Game Design

The fundamental design underpinning the architecture can be summarised as follows:

Each Player has to enrol with the Game Controller, at which point the Game Controller will send the Player the state of the current game, in which they can now participate.

In this version Players can only see the cards they have turned over. They cannot observe the cards turned over by other Players. That challenge is left to the final chapter of the book.

A Player is presented with a graphical representation of the game showing the cards that have yet to be claimed and also the names of the other Players and the number of pairs they have claimed since joining the game.

A Player chooses two cards by pressing the mouse over each of the two cards they wants to reveal. Players can reveal cards in their own time independently of the other players.

If the cards match, the Player implicitly makes a Claim for that pair, otherwise an interface button is enabled to allow the player to replace the cards and then select another pair of cards.

The Claim may not be successful even if it were a pair because another Player may have claimed that pair previously. After each Claim, successful or not, the state of the game the player can see is updated to the current state. The game state includes both the cards that are yet to be matched and also the number of pairs each player has successfully claimed.

When all the cards in the current game have been claimed, the Game Controller will create a new game, and send it to each of the enrolled Players.

A Player can withdraw from the game at any time.

The main area of contention in this design occurs when more than one Player claims the same pair. The Game Controller has to be able to deal with this situation, informing the Players which of them were successful.

1.3.2 Player Interface

The Player Interface is shown in Figure 1-2. It is shown in a state where a Player named Jon has enrolled in the game. The player has revealed two cards, which do not match and thus the 'SELECT NEXT PAIR' button has been enabled. Once that button is pressed the revealed cards will revert to the 'greyed out' state and the player will be able to reveal more cards.

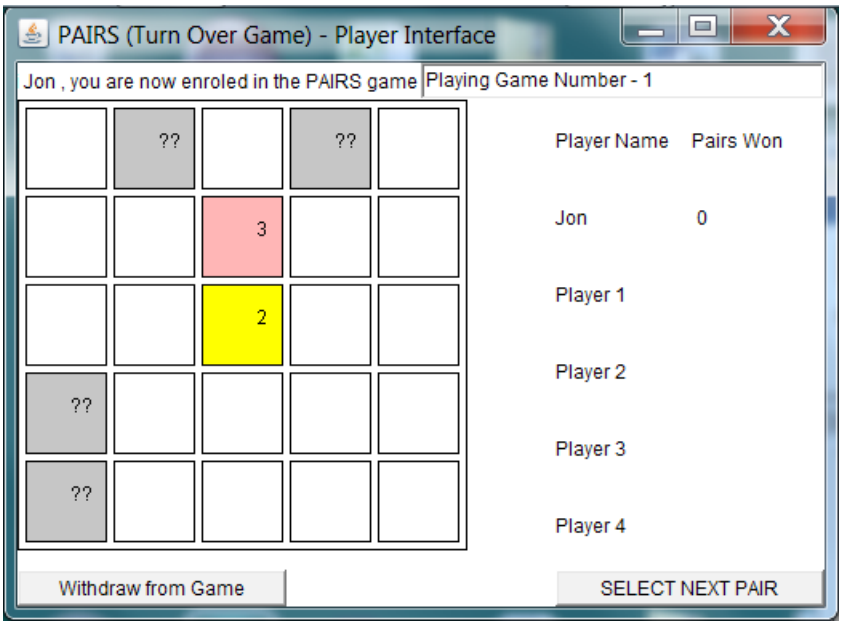


Figure 1-2 The Player Interface

1.3.3 Messages

Messages can be sent in both directions, from the Player to the Game Controller and vice-versa.

1.3.3.1 From Player to Game Controller

Enrol Player – sends enrolment data, once the player has input required data

Withdraw Player – sends a request to withdraw the player from the game

Get Game Details – a request for the controller to send the current Game Details

Claim Pair – contains details of a matched pair of cards that the Player is claiming

1.3.3.2 From Controller to Player

Enrol Details – details to update the names of the players

Game Details – the current state of the game including available cards and player scores

1.3.4 Player Internal Structure

The Player contains a user interface that comprises two parts. The first, concerns the creation of the graphical output and the second deals with events such as button and mouse presses and textual input. The design process means we can deal with these separately because, as we shall see in Chapter 11, each element of a user interface has its own process based architecture which means that such components can be integrated more easily into a parallel system design. Thus when a button is pressed it communicates a message that contains the current text associated with the button. The text in a button can be changed by sending a text string to the button process. Mouse events are dealt with in a similar manner in that a mouse event process communicates the mouse events so they can be processed by another process.

A first design for the Player might look something like that shown in Figure 1-3.

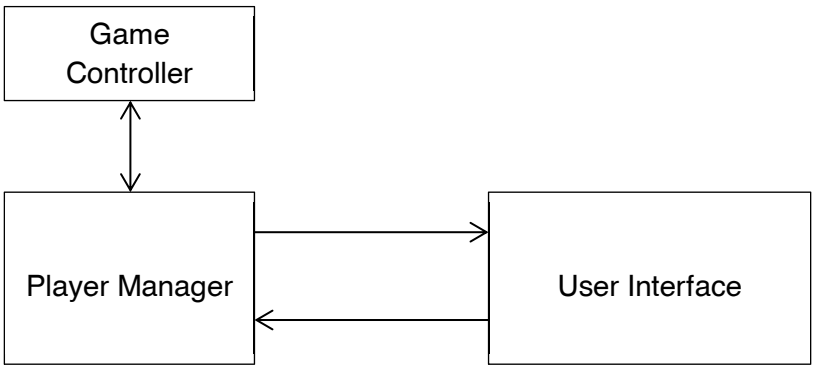


Figure 1-3 A First Player Design

In this design we recognise the fact that there are two primary Player components the Player Manager and a User interface and that we send messages between these components.

The Player Manager deals with interactions between the Player and the Game Controller, while at the same time dealing with interactions from the User Interface. This is a reasonable design until we consider the effect of Mouse Events on the Player Manager. Mouse Events happen in bursts as the mouse is moved or pressed and further the number of events generated is large, most of which are not significant to the operation of the system. We are only interested in mouse presses that occur in a greyed out square. This in the next stage in the design; mouse events are separated from the other interactions with the User Interface, such as Label and Button processing and the update of the paintable Canvas that holds the representation of the cards.

Figure 1-4 shows the next design, where a Mouse Event Buffer has been introduced. This is sent all mouse events but then filters these so that only mouse presses from the card canvas are retained and then passed to the Player Manager.

This is an improvement but still does not provide a completely satisfactory solution. The Player Manager really only wants to deal with mouse presses that are contained within a square that is currently greyed out. This processing can be placed in a separate process. The Player Manager then simply has to determine whether a pair of cards matches, deal with update of the interface and possibly claiming a matched pair of cards.

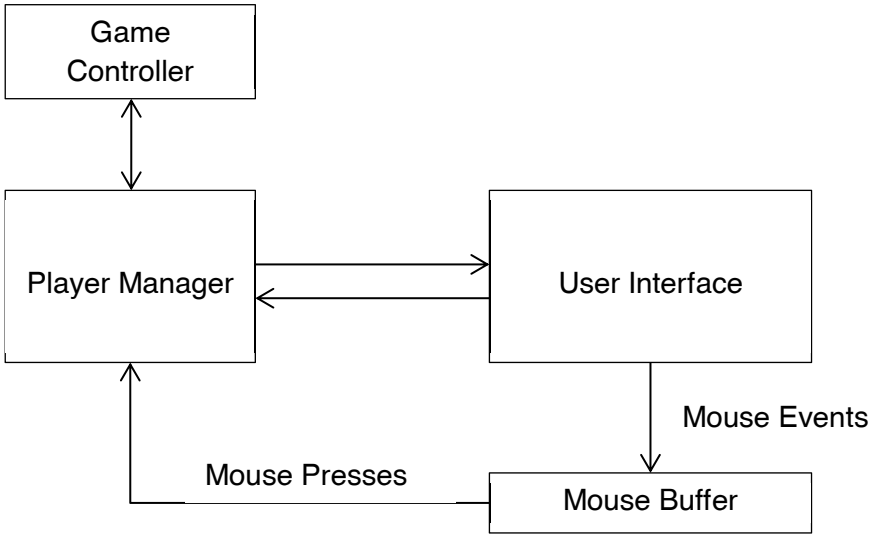


Figure 1-4 Separation of Mouse Events

The revised design is shown in Figure 1-5. The Matcher simply determines whether the mouse press is within a square that is currently greyed out.

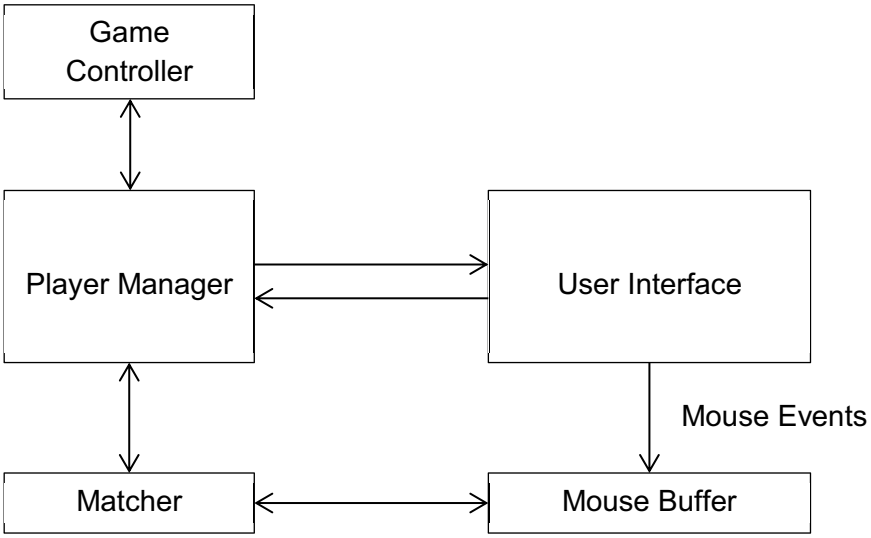


Figure 1-5 Inclusion of Matcher Processing

A further refinement has also been introduced as indicated by the bi-directional connection between the Player Manager and the Matcher and the Matcher and the Mouse Buffer. The Player Manager will know when it needs another point either to be the first or second in a pair. Thus it can ask the Matcher for a valid point, which it then can input and process. Similarly the Matcher can only deal with a press at specific points in the algorithm so it requests a mouse press event from the Mouse Buffer which it then inputs and processes. This refinement, may at first sight appear to complicate the design, however, as we shall see in Chapter 8, this implements a specific design pattern that we shall use throughout the remainder of the book. This so-called client-server design is fundamental to the design of parallel systems that are deadlock and livelock free.

1.3.5 Player Manager Communications

Initially, the only action the User Interface permits is to enable the user to input their name and also to type in the IP-address of the Game Controller. The latter enables the dynamic connection of the Player to the Game Controller and the former enables the Player to enrol in the game. The Game Controller responds with the list of players currently enrolled in the game with the new player's name added to the list and the number of claimed pairs set to zero.

Once enrolled, the Player can withdraw from the game at any time. The problem is the time when this withdrawal occurs is not known. Thus the Player Manager has to be able to deal with it at any time. This is referred to as non-deterministic behaviour in that it is known when an event can occur but not when. The language architecture includes a specific language structure that captures this type of behaviour and is known as a non-deterministic choice. It is fundamental to the design of parallel systems and will be described further in Chapter 3.

The rest of the behaviour is relatively straightforward, once we are aware of the possibility of a withdraw event is non-deterministic and can build that into the design. The Player Manager can request the co-ordinates of a card from the Matcher. The Matcher will respond as and when it has a valid card. Once the Player Manager has two valid cards it can check to see if they correspond. If they do it can send a Claim message to the Game Controller, which will respond with the updated game state including the number of pairs each player has achieved.

If the pair of cards does not correspond, then the Player Manager can enable the 'SELECT NEXT PAIR' button on the User Interface. Only when that button is pressed by the player will the cards return to the greyed out state and the player will be able to select another pair of cards.

1.3.6 Summary

The above description has glossed over a number of important ideas, which will be explored in more detail later on in the book. It has, however, introduced the basic fundamentals that concern the design and implementation of parallel systems. This is that we pass messages between processing elements hence the concept of Communicating Process Architectures. The designer has to manage these interactions and that we have a very small number of design principles that need to be followed to build functional, correct and maintainable parallel systems.

1.4 The Basic Concepts

The fundamental concepts that we shall be dealing with, when designing and thinking parallel are:

Process,
Channel,
Timer,
Alternative.

In comparison to other concurrent and parallel based approaches, the list is very small but that is because we are dealing with higher-level concepts and abstractions. Therefore it is much easier for the programmer to both build concurrent and parallel systems and also to reason about their behaviour. One of the key aspects of this style of parallel system design is that processes can be composed into larger networks of processes with a predictable overall behaviour.

1.4.1 Process

A process, in its simplest form, defines a sequence of instructions that are to be carried out. Typically, a process will communicate with one or more processes using a channel to transfer data from one to the other. In this way a network of processes collectively provide a solution to some problem. Processes have only one method, `run()`, which is used to invoke the process. A list of process instances is passed to an instance of a `PAR` object which, when `run`, causes the parallel execution of all the processes in the list (Kerridge, et al., 2005). A `PAR` object only terminates when all the processes in the list have themselves terminated.

A process encapsulates the data upon which it operates. Such data can only be communicated to or from another process and hence all data is private to a process. Although a process definition is contained within a `Class` definition, there are no explicit methods by which any property or attribute of the process can be accessed.

A network of processes can be invoked concurrently on the same processor, in which case the processor is said to interleave the operations of the processes. The processor can actually only execute one process at a time and thus the processor resource is shared amongst the concurrent processes.

A network of processes can be created that runs on many processors connected by some form of communication mechanism, such as a TCP/IP based network. In this case the processes on the different processors can genuinely execute at the same time and thus are said to run in parallel. In this case some of the processors may invoke more than one process and so an individual processor may have some processes running concurrently but the complete system is running in parallel. The definition of a process remains the same regardless of whether it is executed concurrently or in parallel. Furthermore the designer does not have to be aware, when the process is defined, whether it will execute concurrently or in parallel.

A network of processes can be run in parallel on a multi-core processor in such a way that the processes are executed on different cores. We can thus exploit multi-core processors directly by the use of a process based programming environment. The exploitation of multi-core processors will result in those processes running on the same core executing concurrently and those on different cores in parallel.

Throughout the rest of this book we shall refer to a network of parallel processes without specifically stating whether the system is running concurrently or in parallel. Only when absolutely necessary will this be differentiated.

1.4.2 Channel

A channel is the means by which a process communicates with another process. A channel is a one-way, point-to-point, unbuffered connection between two processes. One process `write`s to the channel and the other `read`s from the channel. Channels are used to transfer data from the outputting (writing) process to the inputting (reading) process. If we need to pass data between two processes in both directions then we have to supply two channels, one in each direction. Channels synchronise the processes to pass data from one to the other. Whichever process attempts to communicate first waits, idle, using no processor resource until the other process is ready to communicate. The second process attempting to communicate will discover this situation, undertake the data transfer and then both processes will continue in parallel, or concurrently if they were both executed on the same processor. It does not matter whether the inputting or outputting process attempts to communicate first the behaviour is symmetrical. At no point in a channel communication interaction does one process cycle round a loop determining whether the other process is ready to communicate. The implementation uses neither polling nor busy-wait-loops and thus does not incur any processor overhead.

This describes the fundamental channel communication mechanism; however, within the parallel environment it is possible to create channels that have many reader and / or writer processes connected to them. In this case the semantics are a little more complex but in the final analysis the communication behaves as if it were a one-to-one communication.

When passing data between processes over a channel some care is needed because, in the Java and Groovy environment, this will be achieved by passing an object reference if both processes are executing concurrently on the same processor. In order that neither of the processes can interfere with the behaviour of each other we have to ensure that a process does not modify an object once it has been communicated. This can be most easily achieved by always defining a new instance of the object which the receiving process can safely modify.

If the communication is between processes on different processors this requirement is relaxed because the underlying system has to make a copy of the data object in any case. An object reference has no validity when sent to another processor. Such a data object has to implement the `Serializable` interface.

If the processes are running on a multi-core processor then they should be treated as processes running concurrently on the same processor because such processes can share the same caches and thus processes will be able to access the same object reference.

1.4.3 Timers

A key aspect of the real world is that many systems rely on some aspect of time, either absolute or relative. Timers are a fundamental component of a parallel programming environment together with a set of operations. Time is derived from the processor's system clock and has millisecond accuracy. Operations permit the time to be read as an absolute value. For example, processes can be made to go idle for some defined, sleep, period. Alarms can be set, for some future time, and detected so that actions can be scheduled. A process that calls the `sleep()` method or is waiting for an alarm is idle and consumes no processor resource until it resumes execution.

1.4.4 Alternatives

The real world in which we interact is non-deterministic, which means that the specific ordering of external events and communications cannot be predefined in all cases. The programming environment therefore has to reflect this situation and permit the programmer to capture such behaviour. The alternative captures this behaviour and permits selection between one or more input communications, timer alarms and other synchronisation capabilities. The events over which the alternative makes its selection are referred to as guards. If one of the guards is ready then that one is chosen and its associated process carried out. If none of the guards are ready then the alternative waits, doing nothing, consuming no processor resource until one is ready. If more than one is ready, it chooses one of the ready guards according to some selection criterion. The ability to select a ready guard is a crucial requirement of any parallel programming environment that is going to model the non-deterministic real world.

1.5 Summary

This brief chapter has defined the terms we are going to use during the rest of the book. From these basic concepts we are going to build many example concurrent and parallel systems simply by constructing networks of processes, connected by channels, each contributing, in part, to the solution of a problem. Whether the network of processes is run in parallel over a network, in a multi-core processor, or concurrently on a single processor has no bearing upon the design of the system. In some systems, the use of multiple processors may be determined by the nature of the external system and environment to which the computer system is connected.

2 Producer Consumer: A Fundamental Design Pattern

This chapter provides an introduction to

- a simple producer – consumer design pattern
- shows how a set of processes can be invoked using the PAR helper class
- shows how processes and channels interact with one another
- demonstrates the ease with which processes can be reused

For many people, the first program they write in a new language is to print "Hello World", followed by the inputting of a person's name so the program can be extended to print "Hello *name*". In the parallel world this is modified to a program that captures one of the fundamental design patterns of parallel systems, namely a Producer – Consumer system.

A Producer process is one that outputs a sequence of distinct data values. A Consumer process is one that inputs a stream of such data values and then processes them in some way. The composition of a Producer and Consumer together immediately generate some interesting possibilities for things to go wrong. What happens if?

the Producer process is ready to output some data before the Consumer is ready or

the Consumer process is ready to input but no data is available from the Producer

In many situations, the programmer would resort to introducing some form of buffer between the Producer and Consumer to take account of any variation in the execution rate of the processes. This then introduces further difficulties in our ability to reason about the operation of the combined system; such as the buffer becomes full so the Producer has to stop outputting, or conversely it becomes empty and the Consumer cannot input any data. We have just put off the decision. In fact, we have made it much harder to both program the system and to reason about it. In addition, we now have to consider the situation when the buffer fails for some reason. Fortunately, the definitions of process and channel given in Chapter 1 come to our rescue.

If the Producer process is connected to the Consumer process by a channel then we know that the processes synchronise with each other when they transfer data over the channel. Thus if the Producer tries to output or write data before the Consumer is ready to input or read data then the Producer waits until the Consumer is ready and vice-versa. It is therefore impossible for any data to be lost or spurious values created during the data communication.

2.1 A Parallel Hello World

2.1.1 Hello World Producer

The producer process for Hello-World is shown in Listing 2-1. Line {10–20} defines the class `ProduceHW` that implements the interface `CSProcess`, which defines a single method `run()` that is used to invoke the process. The interface `CSProcess` is contained in the package `org.jcsp.lang`, which has to be imported (not shown).

```
10 class ProduceHW implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run() {
15     def hi = "Hello"
16     def thing = "World"
17     outChannel.write ( hi )
18     outChannel.write ( thing )
19   }
20 }
```

Listing 2-1 Hello World Producer Process

The only class property, `outChannel` {12}, of type `ChannelOutput`, is the channel upon which the process will output using a `write()` method. Strictly, Groovy does not require the typing of properties, or any other defined variable, however, for documentation purposes we adopt the convention of giving the type of all properties. This also has the advantage of allowing the compiler to check type rules and provides additional safety when processes are formed into process networks. Each process has only one method, the `run()` method as shown starting at line {14}. Two variables are defined {15, 16}, `hi` and `thing`, that hold the strings "Hello" and "World" respectively. These are then written in sequence to `outChannel` {17, 18}.

2.1.2 Hello World Consumer

The `ConsumeHello` process, Listing 2-2, has a property `inChannel` {12} of type `ChannelInput`. Such channels can only input objects from the channel using a `read()` method. Its `run()` method firstly, reads in two variables, `first` and `second`, from its `inChannel` {15, 16}, which are then printed {17} to the console window with preceding and following new lines (\n). The notation `$v` indicates that the variable `v` should be evaluated to its `String` representation

```
10 class ConsumeHello implements CSProcess {
11
12   def ChannelInput inChannel
13
14   void run() {
```

```
15      def first = inChannel.read()
16      def second = inChannel.read()
17      println "\n$first $second!\n"
18    }
19 }
```

Listing 2-2 Hello World Consumer Process

2.1.3 Hello World Script

Figure 2-1 shows the process network diagram for this simple system comprising two processes `ProduceHW` and `ConsumeHello` that communicate over a channel named `connect`.

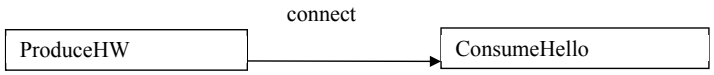


Figure 2-1 Producer Consumer Process Network

The script, called `RunHelloWorld` for executing the processes `ProduceHW` and `ConsumeHW` is shown in Listing 2-3. It is this script that is invoked to execute the process network.

```
10 def connect = Channel.one2one()
11
12 def processList = [
13                       new ProduceHW ( outChannel: connect.out() ),
14                       new ConsumeHello ( inChannel: connect.in() )
15                   ]
16 new PAR (processList).run()
```

Listing 2-3 Hello World Script

An imported package `org.jcsp.lang` contains the classes required for the JCSP library. Another package `org.jcsp.groovy` contains the definitions of the Groovy parallel helper classes. The referenced libraries and documentation contain a more complete specification and description of their use. The `PAR` class {16} causes the parallel invocation of a list of processes. This is achieved by calling the `run()` method of `PAR`, which in turn causes the execution of the `run()` method of each of the processes in the list.

The channel `connect` is of type `Channel.one2one` {10}. The channel is created by means of a static method `one2one` in the class `Channel` contained within the package `org.jcsp.lang`. The `processList` {12} comprises an instance of `ProducerHW` with it's `outChannel` property set to the `out` end of `connect`, and the processes `ConsumerHW` with its `inChannel` property set to the `in` end of `connect` {13, 14}.

The underlying JCSP library attempts, as far as possible, to ensure networks of processes are connected in a manner that can be easily checked. The channel `connect` {10} is defined to have a `one2one` interface and therefore it has one output end and one input end. These are defined by the methods `out()` {13} and `in()` {14} respectively. A class that contains a property of type `ChannelOutput` must be passed an actual parameter that has been defined with a call to `out()` and within that process only `write()` method calls are permitted on the channel property. The converse is true for input channels. In all process network diagrams the arrow head associated with a channel will refer to the input end of the channel.

The output from executing this script is shown in Output 2-1.

```
Hello World!
```

Output 2-1 Output from Hello World Script

2.2 Hello Name

The Hello Name system is a simple extension of the Hello World system. The only change is that the `ProducerHN` {10} process asks the user for their name and then sends this to the `Consumer` process as the `thing` variable {16, 18} in Listing 2-4.

```
10 class ProduceHN implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run() {
15     def hi = "Hello"
16     def thing = Ask.string ("\nName ? ")
17     outChannel.write ( hi )
18     outChannel.write ( thing )
19   }
20 }
```

Listing 2-4 The ProduceHN Process

An imported package `phw.util` contains some simple console interaction methods that can be used to obtain input from the user from the console window. The `Ask.string` method outputs the prompt "`Name ?`" after a new line and the user response is then placed into the variable `thing` {16}.

The Consumer process remains unaltered from the version shown in Listing 2-2. Similarly, the script to run the processes is the same as Listing 2-3 except that the name of the producer process has been changed to `ProduceHN`. A typical output from the execution of the script is shown in Output 2-2, where user typed input is shown in *italics*. This also shows how easy it is to reuse a process in another network.

```
Name ? Jon
Hello Jon!
```

Output 2-2 Output from Hello Name Network

2.3 Processing Simple Streams of Data

The final example in this chapter requires the user to type in a stream of integers into a producer process that writes them to a consumer process, which then prints them. The specification of the `Producer` process is given in Listing 2-5.

```
10 class Producer implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run() {
15     def i = 1000
16     while ( i > 0 ) {
17       i = Ask.Int ("next: ", -100, 100)
18       outChannel.write (i)
19     }
20   }
21 }
```

Listing 2-5 The Producer Process

The `run()` {14} method is formulated as a `while` loop {16–19}, which is terminated as soon as the user inputs zero or negative number. The input integer value is obtained using the `Ask.Int` (from phw.util) method that will ensure that any input lies between -100 to 100 {17}. The `while` loop has been structured to ensure the final zero is also output to the `Consumer` process.

```
10 class Consumer implements CSProcess {
11
12   def ChannelInput inChannel
13
14   void run() {
15     def i = 1000
16     while ( i > 0 ) {
17       i = inChannel.read()
18       println "the input was : $i"
19     }
20     println "Finished"
21   }
22 }
```

Listing 2-6 The Consumer Process

The Consumer process is shown in Listing 2-6. The `Consumer` {10} process reads data {17} from its input channel, `inChannel` {12}, which is then printed {18}. Once a zero is read the `while` loop {16–19} terminates resulting in the printing of the "`Finished`" message {20}.

The script, called `RunProducerConsumer` that causes the execution of the network of processes is shown in Listing 2-7 which is very similar to the previous script shown in Listing 2-3, the only change being, the names of the processes that make up the list of processes {12, 13}.

```
10  def connect = Channel.one2one()
11
12  def processList = [ new Producer ( outChannel: connect.out() ),
13                      new Consumer ( inChannel: connect.in() )
14                    ]
15  new PAR (processList).run()
```

Listing 2-7 The Producer Consumer System Script

Output from a typical execution of the processes is given in Output 2-3.

```
next: 1
next: the input was : 1
2
the input was : 2
next: 3
the input was : 3
next: 0
the input was : 0
Finished
```

Output 2-3 Typical Results from Producer Consumer System

The output, especially when executed from within Eclipse, can be seen to be correct but the output is somewhat confused as we have two processes writing concurrently to a single console at the same time; both processes use `println` statements. We therefore have no control of the order in which outputs appear and these often become interleaved. A process called GConsole is available in the package `org.jcsp.groovy.plugAndPlay` that creates a console window with both an input and output area. This process can be used to provide a specific console facility for a given process. Many such GConsole processes can be executed in a network of processes as required. Its use will be demonstrated in later chapters.

2.4 Summary

This chapter has introduced the basic use of many simple JCSP concepts. A set of simple Producer – Consumer based systems have been implemented and output from these systems has also been given. These basic building blocks of processes and channels, with their simple semantics are the basis for all the concurrent and parallel systems we shall be building throughout the rest of this book.

2.5 Exercises

Exercise 2-1

Using Listing 2-7 as a basic design implement and test a new process called `Multiply` that is inserted into the network between the `Producer` and `Consumer` processes which takes an input value from the `Producer` process and multiplies it by some constant `factor` before outputting it to the `Consumer` process. The multiplication `factor` should be one of the properties of the `Multiply` process. To make the output more meaningful you may want to change the output text in the `Consumer` process. You should reuse the `Producer` process from the `ChapterExamples` project in `src` package `c2`.

Exercise 2-2

A system inputs data objects that contain three integers; it is required to output the data in objects that contain eight integers. Write and test a system that undertakes this operation. The process `ChapterExercises/src/c2.GenerateSetsOfThree` outputs a sequence of `Lists`, each of which contains three positive integers. The sequence is to be terminated by a `List` comprising [-1, -1, -1].

What change is required to output objects containing six integers? How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12)? What happens if the number of integers required in the output stream is not a factor of the total number of integers in the input stream (e.g. 5 or 7)?

3 Process Networks: Build It Like Lego

We now progress to more complex networks

- using simple and easily understood processes to build larger networks
- the reuse of existing processes is demonstrated
- a timer is used to improve console readability
- introduces the concepts of ChannelInputLists and ChannelOutputLists

One of the main advantages of the CSP based approach we are using is that processes can be combined using a simple compositional style. It is very much *what you see is what you get*!

In arithmetic the meaning of the composition 1 + 2 + 3 is immediately obvious and results in the answer 6. The composition of processes is equally simple and obvious. Thus we can build a set of basic building block processes, like Lego® bricks, from which we can construct larger systems, the meaning of which will be obvious given our understanding of the basic processes.

All of the building block processes are to be found in the package `org.jcsp.groovy.plugAndPlay`. A more detailed discussion of these processes is to be found in the accompanying documentation.

3.1 Prefix Process

The process diagram of `GPrefix` is given in Figure 3-1 and its definition is presented in Listing 3-1. `GPrefix` initially outputs the `prefixValue` on its `outChannel` {17} and thereafter it writes everything it reads on its `inChannel` {13} to its `outChannel`, using a non-terminating loop {18-19}.

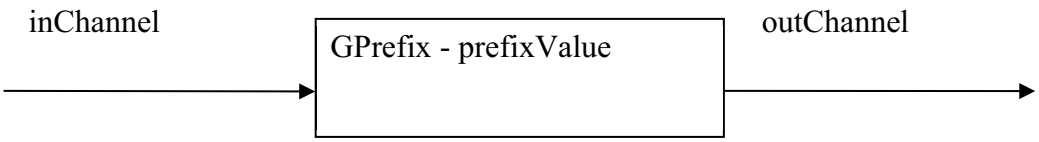


Figure 3-1 GPrefix Process Diagram

The `GPrefix` process has an input channel `inChannel` and an output channel `outChannel`, which are properties of the process {13, 14}. In addition, there is a property called `prefixValue` that has the initial value 0 {12}, which can be changed when a process instance is created.

```
10 class GPrefix implements CSProcess {
11
12   def int prefixValue = 0
13   def ChannelInput inChannel
14   def ChannelOutput outChannel
15
16   void run () {
17     outChannel.write(prefixValue)
18     while (true) {
19       outChannel.write( inChannel.read() )
20     }
21   }
22 }
```

Listing 3-1 GPrefix Process Definition

3.2 Successor Process

The process diagram for `GSuccessor` is shown in Figure 3-2 and it's coding in Listing 3-2. The process simply {17} reads in a value from its `inChannel` and then writes this value plus 1 to its `outChannel`. It does this in a `while` loop that never terminates {16-17}.

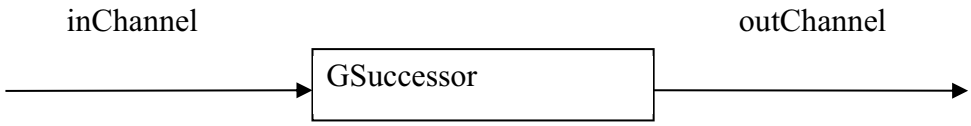


Figure 3-2 GSuccessor Process Diagram

```
10 class GSuccessor implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14
15   void run () {
16     while (true) {
17       outChannel.write( inChannel.read() + 1 )
18     }
19   }
20 }
```

Listing 3-2 GSuccessor Process Definition

3.3 Parallel Copy

The process diagram for `GPCopy` is given in Figure 3-3 and it's coding in Listing 3-3. The process inputs a value on its `inChannel` {12}, which it outputs to `outChannel0` {13} and `outChannel1` {14} in parallel. This is repeated forever. By outputting to its output channels in parallel we are assured that it does not matter the order in which these channels are read by other processes on their corresponding input channels. We are also guaranteed that a read on its input channel will not take place until both the outputs have completed because a parallel (PAR) does not terminate until all its constituent processes have terminated.

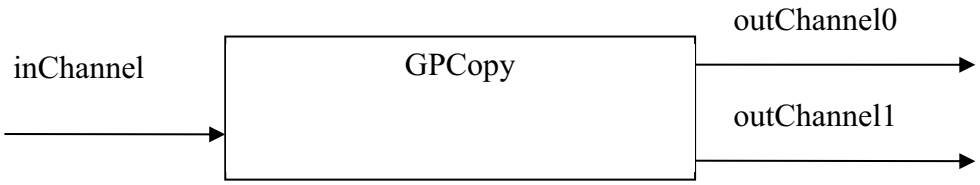


Figure 3-3 Process Diagram of GPCopy

`GPCopy` utilises the process `ProcessWrite` from the package `org.jcsp.plugNplay`, demonstrating that we can incorporate previously written Java processes into the Groovy environment. Two instances of `ProcessWrite` are defined {17, 18} each accessing one of the output channels. A `PAR` of the two processes is then defined {19} called `parWrite2`, which is not run at this time. An instance of `ProcessWrite` has a publicly available field called `value` that is assigned the data to be written.

The non-terminating loop {20–25} firstly reads in a value from the `inChannel` {21}, the value of which is assigned to the `value` fields of the two `ProcessWrite` instances, `write0` {22}and `write1` {23}. The parallel `parWrite2` is then run {24}, which causes the writing of the value read in from `inChannel` to `outChannel0` and `outChannel1` in parallel, after which it terminates. `ProcessWrite` terminates as soon as it has written a single value to its output channel. Once `parWrite2` has terminated, processing resumes at the start of the `while` loop {20}.

```
10 class GPCopy implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel0
14   def ChannelOutput outChannel1
15
16   void run () {
17     def write0 = new ProcessWrite ( outChannel0)
18     def write1 = new ProcessWrite ( outChannel1)
19     def parWrite2 = new PAR ( [ write0, write1 ] )
20     while (true) {
21       def i = inChannel.read()
22       write0.value = i
23       write1.value = i
24       parWrite2.run()
25     }
26   }
27 }
```

Listing 3-3 GPCopy Process Definition

3.4 Generating a Sequence of Integers

The three processes, `GPrefix`, `GSuccessor` and `GPCopy` can be combined to form a network that outputs a sequence of integers on `outChannel` as shown in Figure 3-4 and Listing 3-4.

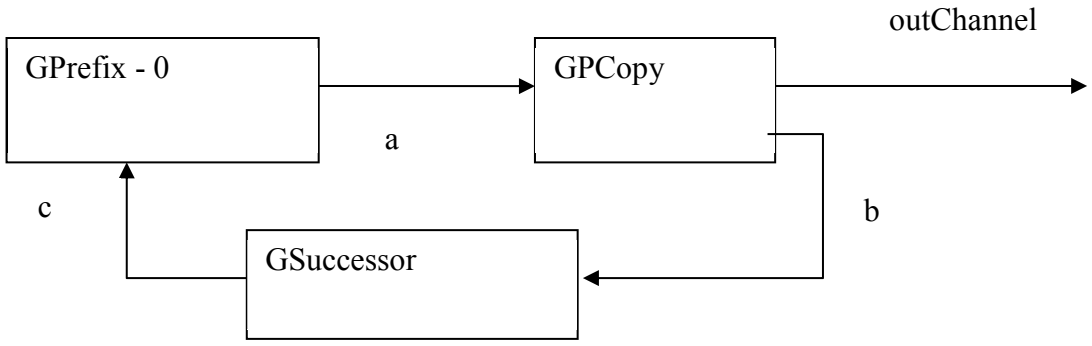


Figure 3-4 Process Network Diagram to Generate a Stream of Integers (GNumbers)

The operation of the network proceeds as follows. Initially, only `GPrefix` can undertake any processing which is to output its prefix value, 0. This is then input by `GPCopy`, which copies the value to both the `outChannel` and the input of `GSuccessor`. `GSuccessor` then reads the input value, increments it and then outputs it to `GPrefix`. `GPrefix` then copies the new input value to its output channel. Thus the numbers circulate round the network incrementing by one each time.

The `GNumbers` process has a single output channel `outChannel` property {12} upon which the stream of integers is output. Three internal channels `a`, `b` and `c` are defined {16–18} as `one2one` channels and these are used to connect the processes together in a manner that directly reflects the process network diagram, Figure 3-4. For example, the two output channels of `GPCopy` are assigned to the property `outChannel` and `b.out()` while its input channel is assigned to `a.in()`.

```
10 class GNumbers implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run() {
15
16     def a = Channel.one2one()
17     def b = Channel.one2one()
18     def c = Channel.one2one()
19
```

```
20      def numbersList = [ new GPrefix ( prefixValue: 0,
21                                        inChannel: c.in(),
22                                        outChannel: a.out() ),
23                          new GPCopy ( inChannel: a.in(),
24                                       outChannel0: outChannel,
25                                       outChannel1: b.out() ),
26                          new GSuccessor ( inChannel: b.in(),
27                                           outChannel: c.out())
28                        ]
29      new PAR ( numbersList ).run()
30    }
31 }
```

Listing 3-4 Definition of the GNumbers Process

The design process becomes one of creating a process network diagram and then using that to define the required channels which are then used to connect the processes together. The system is able to check, using the interface specifications, that an input end of a channel specified by the `in()` method is connected to a `ChannelInput`. Similarly a `ChannelOutput` must be connected to a channel output end specified by the `out()` method because we have specified the types of the channels in the properties of the process class definitions.

3.5 Testing GNumbers

Figure 3-5 shows the process network that can be used to test the operation of the process `GNumbers`. It is apparent that the easiest way of testing the process `GNumbers` is to print the stream of numbers to the console window. For this purpose a `GPrint` process is provided. `GPrint` has a `ChannelInput` for reading a stream of numbers from its `inChannel` property. It also has a property, `heading`, that is a `String`, which contains a title for the printed stream. The corresponding script for the network shown in Figure 3-5 is given in Listing 3-5.

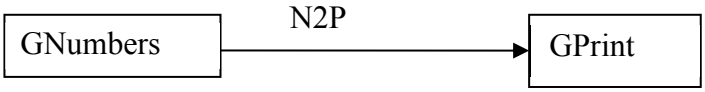


Figure 3-5 Network to Test GNumbers

```
10 def N2P = Channel.one2one()
11
12 def testList = [ new GNumbers ( outChannel: N2P.out() ),
13                  new GPrint ( inChannel: N2P.in(),
14                               heading : "Numbers" )
15           ]
16 new PAR ( testList ).run()
```

Listing 3-5 The Script to Test GNumbers

A single channel is created {10} called `N2P` that is used to connect `GNumbers` to `GPrint`. The list of processes is created {12–15} with the properties assigned to the input and output ends of `N2P` and the `heading` property of `GPrint` is set to "Numbers". A typical output is shown in Output 3-1. It is noted that the user has to terminate the system by interrupting the console stream. The processes have been constructed using never ending while-loops and thus run forever, unless otherwise terminated. In Eclipse this is achieved by clicking the 'Terminate' button in the Console view.

```
Numbers
0
1
2
3
4
5
6
7
8
9
10
11
```

Output 3-1 Output from the Script Test GNumbers

3.6 Creating a Running Sum

We will now use the output from `GNumbers` as input to a process called `GIntegrate` that reads a stream of integers and outputs the running sum of the numbers read so far, as another stream of numbers. In order to do this we shall need a process that undertakes addition of numbers arriving in a stream of such numbers. The `GPlus` process does this and its coding is shown in Listing 3-6.

```
10 class GPlus implements CSProcess {
11
12   def ChannelOutput outChannel
13   def ChannelInput inChannel0
14   def ChannelInput inChannel1
15
16   void run () {
17
18     ProcessRead read0 = new ProcessRead ( inChannel0)
19     ProcessRead read1 = new ProcessRead ( inChannel1)
20     def parRead2 = new PAR ( [ read0, read1 ] )
21
22     while (true) {
```

```
23          parRead2.run()
24          outChannel.write(read0.value + read1.value)
25        }
26      }
27 }
```

Listing 3-6 GPlus process coding

The `GPlus` process uses techniques similar to that used in `GPCopy`, except that we read from two input channels in parallel using the process `ProcessRead`, which reads a single value from a channel and then terminates. `GPlus` has two input channels, `inChannel0` and `inChannel1` {13, 14} and one output channel, `outChannel` {12} upon which the sum of the two inputs are written. Two `ProcessRead` processes are constructed called `read0` {18} and `read1` {19} and these are used to construct a `PAR` called `parRead2` {20}. It should be noted that {20} only defines the parallel `parRead2`, it does not cause it to be run. The main loop of the process {22-25} initially invokes the parallel `parRead2` {23}. This parallel only terminates when both `read0` and `read1` have read a value and terminated. The values read are obtained from a publicly available field, `value`, of a `ProcessRead`. The two values that have been read are added together and then written to the output channel {24}.

Listing 3-7 gives the coding for the process `GIntegrate` and its associated process network diagram is given in Figure 3-6. The coding can be seen to be a representation of the diagram in the same way as previous transformations of diagrams into process network scripts.

The operation of `GIntegrate` proceeds as follows. The process `GPrefix` can output its initial value, 0, which forms one of the inputs to `GPlus`, using channel `c`. The other input from `GPlus` is read from `GIntegrate`'s `inChannel`. The `GPlus` process and hence the `GIntegrate` process will now wait until there is an input on the `inChannel`. Once this arrives the addition of the two values will take place and the result written to the channel `a`, which forms the input to `GPCopy`. `GCopy` can now output the current sum on the `outChannel` and also send a copy to `GPrefix`, using channel `b`, which immediately outputs the value unaltered to the channel `c`. In this way the current running sum is circulated around the network and is also output to a subsequent process.

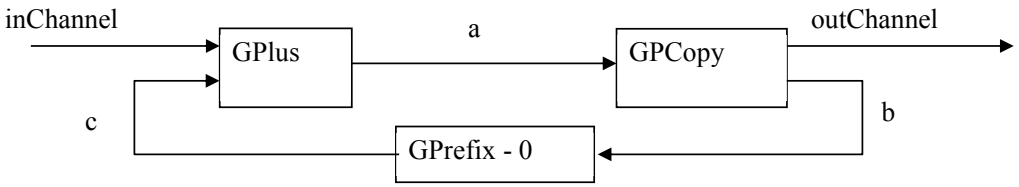


Figure 3-6 Process Network Diagram of GIntegrate

```
10 class GIntegrate implements CSProcess {
11
12   def ChannelOutput outChannel
13   def ChannelInput inChannel
14
15   void run() {
16
17     def a = Channel.one2one()
18     def b = Channel.one2one()
19     def c = Channel.one2one()
20
21     def integrateList = [ new GPrefix ( prefixValue: 0,
22                                         outChannel: c.out(),
23                                         inChannel: b.in() ),
24                           new GPCopy ( inChannel: a.in(),
25                                        outChannel0: outChannel,
26                                        outChannel1: b.out() ),
27                           new GPlus ( inChannel0: inChannel,
28                                       inChannel1: c.in(),
29                                       outChannel: a.out() )
30                         ]
31     new PAR ( integrateList ).run()
32
33   }
34 }
```

Listing 3-7 GIntegrate Process Definition

A process network to test the operation of `GIntegrate`, by outputting the current value of the running sum is presented in Figure 3-7. `GNumbers` provides the input stream into `GIntegrate` using the channel `N2I` and the output from `GIntegrate` is written, using the channel `I2P`, to the `GPrint` process which writes the stream of numbers to the console.

The script that invokes this network is shown in Listing 3-8. The script is taken directly from the process network diagram by connecting the output and input ends of each of the channels, `N2I` and `I2P`, to the appropriate property of the processes.

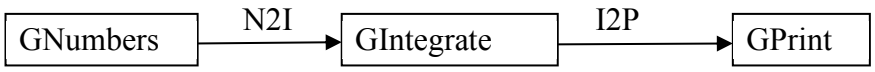


Figure 3-7 The Process Network to Demonstrate the Operation of GIntegrate

```
10 def N2I = Channel.one2one()
11 def I2P = Channel.one2one()
12
13 def testList = [ new GNumbers ( outChannel: N2I.out() ),
14                   new GIntegrate ( inChannel: N2I.in(),
15                                    outChannel: I2P.out() ),
16                   new GPrint ( inChannel: I2P.in(),
17                                heading: "Integrate")
18             ]
19
20 new PAR ( testList ).run()
```

Listing 3-8 Script that Implements the Network of Figure 3-7

Output 3-2 shows the console window after the network has been allowed to execute for a short period of time. It can be seen by observation that each output is the sum of the numbers so far from the sequence 0, 1, 2, …. Later, we shall see how we can output all the intermediate values.

```
Integrate
0
1
3
6
10
15
21
28
36
45
```

Output 3-2 Running Sum Generated by the Sequence of Positive Integers

3.7 Generating the Fibonacci Sequence

The Fibonacci sequence comprises; 0, 1, 1, 2, 3, 5, 8, 13, 21, …f_{n-2}+f_{n-1}, …. The first two numbers in the sequence f_0 and f_1 have to be predefined and are typically set to 0 and 1 but could be any value. It can be seen that we need to create the first two numbers in the sequence and we already have a process, `GPrefix` that achieves this. We now need a process that will read two numbers, in sequence and then output the sum of the pair of numbers. The next iteration will take the second number in the sequence and pair it to the third number that is input, output their sum and so on.

3.7.1 Adding Pairs of Numbers

Listing 3-9 gives the definition of a process that inputs a stream of numbers and outputs another stream which contains the sum of pairs of numbers. The process `GStatePairs` initially reads in two numbers from the input stream, `inChannel`, {17, 18} then, within a loop outputs their sum {20} to `outChannel`, assigns the second number to the first {21}and then reads another number `n2` from `inChannel` {22}.

```
10 class GStatePairs implements CSProcess {
11
12   def ChannelOutput outChannel
13   def ChannelInput inChannel
14
15   void run() {
```

```
16
17      def n1 = inChannel.read()
18      def n2 = inChannel.read()
19      while (true) {
20        outChannel.write ( n1 + n2 )
21        n1 = n2
22        n2 = inChannel.read()
23      }
24    }
25 }
```

Listing 3-9 Process GStatePairs

The process network diagram that implements the generation of the Fibonacci sequence is shown in Figure 3-8 and its associated process definition is shown in Listing 3-10.

Initially, `GPrefix-0` is the only process that can run because it is the only one that can undertake an output. `GPCopy` is waiting for an input as is `GStatePairs`. `GPrefix-1` is trying to output and will not be able to, until `GPrefix-0` reads from its input channel, which it will do once it has written the 0 to `GPCopy`.

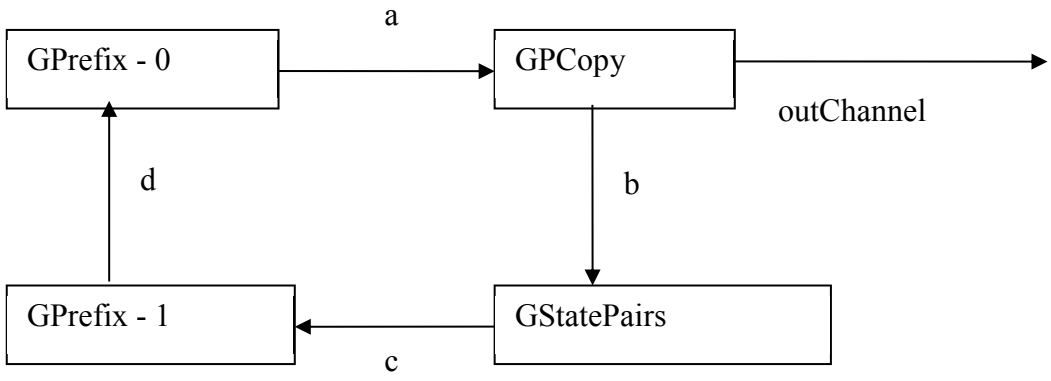


Figure 3-8 Process Network Diagram to Generate the Fibonacci Sequence

It can be seen, by inspection, that the code given in Listing 3-10, directly implements the process network diagram given in Figure 3-8. The four channels, `a`, `b`, `c` and `d` are defined {16–19}. The list of processes is then created as `testList` {21–32} comprising four elements, one for each of the required processes. The list of processes is then invoked using a `PAR` {33}.

```
10 class FibonacciV1 implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run () {
15
16     def a = Channel.one2one()
```

```
17      def b = Channel.one2one()
18      def c = Channel.one2one()
19      def d = Channel.one2one()
20
21      def testList = [ new GPrefix ( prefixValue: 0,
22                                     inChannel: d.in(),
23                                     outChannel: a.out() ),
24                       new GPrefix ( prefixValue: 1,
25                                     inChannel: c.in(),
26                                     outChannel: d.out() ),
27                       new GPCopy ( inChannel: a.in(),
28                                    outChannel0: b.out(),
29                                    outChannel1: outChannel ),
30                       new GStatePairs ( inChannel: b.in(),
31                                         outChannel: c.out() ),
32                     ]
33      new PAR ( testList ).run()
34    }
35 }
```

Listing 3-10 Fibonacci Process Definition

Listing 3-11 shows the script by which the output from the Fibonacci system can be produced using the previously defined `GPrint` process.

```
10 def F2P = Channel.one2one()
11
12 def testList = [ new FibonacciV1 ( outChannel: F2P.out() ),
13                   new GPrint ( inChannel: F2P.in(),
14                                heading: "Fibonacci V1" )
15
16                 ]
17 new PAR ( testList ).run()
```

Listing 3-11 The Script to Output the Fibonacci Sequence

The output from this script is shown in Output 3-3.

```
Fibonacci V1
0
1
1
2
3
5
```

```
8
13
21
34
55
89
```

Output 3-3 Console Output from Script Generating the Fibonacci Sequence

There is, however, a problem with this solution because we now have a process definition for `GStatePairs` (Listing 3-9) that contains some state (`n1` and `n2`) that is retained between iterations of the process. All the other process defined so far, contain no such state. We have also defined a process `GStatePairs` that does addition within it and yet we have already defined a process `GPlus` (Listing 3-6) that undertakes stateless addition. How can we build another process that enables us to reuse the `GPlus` process and which yet can be used to create the effect of `GStatePairs`? This may seem a somewhat esoteric argument but processes that contain state are much more difficult to modify should changes be required in future, especially if it is desired to modfy their behaviour dynamically. This is discussed in the next section.

3.7.2 Using `GPlus` to Create the Sum of Pairs of Numbers

In order to use `GPlus` we need two input streams comprising the numbers to be added together. We can use `GPCopy` to copy the input stream, which would give us two identical streams. We however require adding the current number to the previous one. Hence we require a process that removes the first number from one of the streams and then just outputs what it inputs. If this process is inserted into one of the streams coming from `GPCopy` then we will create that stream with the current number and the other will, in fact contain the previous number. This is shown in Figure 3-9, where the process `GTail` is introduced.

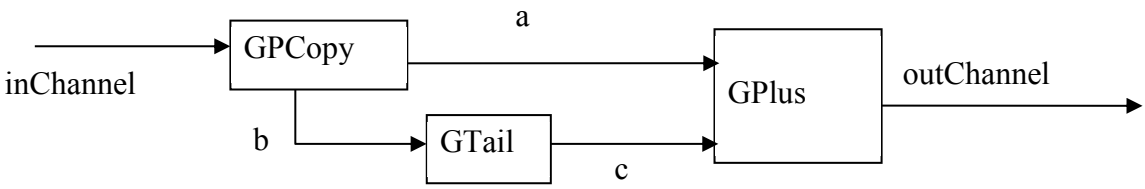


Figure 3-9 GPairs Process Network that Adds Pairs of Numbers using GPlus

The definition of `GTail` is shown in Listing 3-12. The first value sent to `inChannel` is `read` but not retained {16}. Thereafter, values are `read` from `inChannel` and immediately `written` to `outChannel` {14}. This formulation retains no state between iterations of the loop {17–18}.

```
10 class GTail implements CSProcess {
11
12   def ChannelOutput outChannel
13   def ChannelInput inChannel
14
15   void run () {
16     inChannel.read()
17     while (true) {
18       outChannel.write( inChannel.read() )
19     }
20   }
21 }
```

Listing 3-12 Definition of GTail

The operation of the network given in Figure 3-9 is as follows; the first number, 0, is read by `GPCopy` and copied to channels `a` and `b` in parallel. `GTail` reads the 0 on channel `b` and ignores it! Meanwhile the output on channel `a` is read by `GPlus`. `GPCopy` now reads the next number, 1, and attempts to copy this to both channels `a` and `b` in parallel. That to channel `b` will be read by `GTail` and immediately output to channel `c` to be read by `GPlus`, which can now do the addition and subsequent output of the sum of 0 and 1. `GPCopy` is now able to write the copy of 1 to the channel `a` as `GPlus` is now ready to read, in parallel. The system continues in this manner, with none of the processes retaining any state and simply relying on the fact that processes input from and output to multiple channels in parallel and that the order in which the communications takes places does not matter. The semantics of channel communication ensure that no data is lost.

The coding of the stateless version of the process, `GPairs`, to add pairs of numbers from a stream is shown in Listing 3-13 and follows the structure shown in Figure 3-9.

```
10 class GPairs implements CSProcess {
11
12   def ChannelOutput outChannel
13   def ChannelInput inChannel
14
15   void run() {
16
17     def a = Channel.one2one()
18     def b = Channel.one2one()
19     def c = Channel.one2one()
20
21     def pairsList = [ new GPlus ( outChannel: outChannel,
22                                   inChannel0: a.in(),
23                                   inChannel1: c.in() ),
24                       new GPCopy ( inChannel: inChannel,
25                                   outChannel0: a.out(),
26                                   outChannel1: b.out() ),
27                       new GTail ( inChannel: b.in(),
28                                   outChannel: c.out() )
29                       ]
30     new PAR ( pairsList ).run()
31   }
32 }
```

Listing 3-13 The GPairs Process Definition

The definition of the second version of the Fibonnaci process is the same as that given in Listing 3-10 with lines 28–29 replaced with the invocation of the constructor for `GPairs` instead of `GStatePairs`. The execution of the second version is the same as that shown in Listing 3-11 with line 12 creating an instance of the second version of the Fibonnaci process rather than the first. The output is identical from both systems.

3.7.3 Lessons Learned

We should always try to reuse existing processes whenever possible and that often the best way of solving a problem is to define another process rather than changing or extending an existing one. In other words, if we try to keep each process as simple as possible and to compose systems from lots of small, easily understood processes it will be easier to reason about the behaviour of the complete network.

3.8 Generating Squares of Numbers

In this example, we will reuse the processes we have created so far to create a sequence of squares of numbers. The process network to achieve this is shown in Figure 3-10 and the corresponding Listing 3-14 gives the process definition. The process simply writes to it's `outChannel` the squares of the numbers starting with 1 upwards. It can be tested by connecting the `outChannel` to a `GPrint` process.

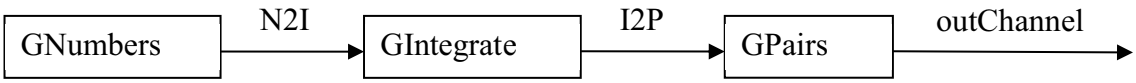


Figure 3-10 The GSquares Process Network

By inspection it can be seen that the `GSquares` process, Listing 3-14, does implement the network given in Figure 3-10. However, what is not obvious is how this result is achieved. To try to understand this we need to print the output from each stage of the squares process. For this we require a process that prints a number of parallel inputs.

```
10 class GSquares implements CSProcess {
11
12   def ChannelOutput outChannel
13
14   void run () {
15
16     def N2I = Channel.one2one()
17     def I2P = Channel.one2one()
18
19     def testList = [ new GNumbers ( outChannel: N2I.out() ),
20                      new GIntegrate ( inChannel: N2I.in(),
21                                       outChannel: I2P.out() ),
22                      new GPairs ( inChannel: I2P.in(),
23                                   outChannel: outChannel ),
24                      ]
25     new PAR ( testList ).run()
26   }
27 }
```

Listing 3-14 GSquares Process Definition

3.9 Printing in Parallel

There are many occasions in which we wish to print output from a set of parallel processes so that the output correlates the state of each process at a consistent point in their execution. The `GParPrint` process achieves this by reading a number of inputs in parallel and then printing out each in a tabular manner one set of inputs to a line of text. Its coding is shown in Listing 3-15.

```
10 class GParPrint implements CSProcess {
11
12   def ChannelInputList inChannels
13   def List<String> headings
14   def long delay = 200
15
16   void run() {
17     def inSize = inChannels.size()
18     def readerList = []
19     (0 ..< inSize).each { i ->
20       readerList [i] = new ProcessRead ( inChannels[i] )
21     }
22
```

```
23      def parRead = new PAR ( readerList )
24
25      if ( headings == null ) {
26        println "No headings provided"
27      }
28      else {
29        headings.each { print "\t$it" }
30        println ()
31      }
32
33      def timer = new CSTimer()
34      while ( true) {
35        parRead.run()
36        readerList.each { pr -> print "\t" + pr.value.toString() }
37        println ()
38        if (delay > 0 ) timer.sleep ( delay)
39      }
40    }
41 }
```

Listing 3-15 The GParPrint Process Defintion

The property `inChannels` {12} is of type `ChannelInputList`, which comprises a list of input channel ends. A `ChannelInputList` is provided as one of the Groovy helper classes in the package `org.jcsp.groovy`. It makes for easier processing of collections of channels. There is a similar object for output channel ends called `ChannelOutputList`. The property `headings` {13} is a `List` of the same size as `inChannels`, though this is not checked, of the title to be placed at the top of each column of printed numbers. The property `delay` {14} is used to introduce a time delay between each line of printed output to make it easier to read as the output appears. The `delay` has a default value of 200 milliseconds and is of type `long` because the system clock returns times in that format. The default value will be used if the property is not assigned a new value when the process is constructed.

The number of `inChannels` in the `ChannelInputList` is obtained by applying the `size()` {17} method. The variable `readerList` is defined {18} as an empty list and will be used to build the list of `ProcessRead` processes that will be used to read from each of the `inChannels` in parallel. The `closure` {19–21} iterates over each element in the range `0` to `inSize-1` and constructs a `ProcessRead` process accessing the `i`'th element of `inChannels` and allocating the instance to the corresponding element of `readerList` {18}. A parallel is then constructed, `parRead`, using `PAR`, from `readerList` {23}. The collection of processes is not executed at this time.

The heading for each column of output is now created {25–31}. If the value of `headings` is `null` {25} then a message indicating that no headings was provided is output {26}. Otherwise a heading is written, tab separated (`\t`) using the elements of the `List headings` by a closure that iterates {29} over the elements of `headings`, using the `each` iterator method. The name `it` refers to the value returned by the iterator. It is assumed but not checked that the number of elements in `headings` is the same as that in `inChannels`.

A `timer` is now defined {33} of type `CSTimer` (see Chapter 9) that will be used to create the delay between each line of output. The main loop of the process can now commence {34–39}. The first requirement is to read the input values in parallel by executing `parRead` {35}. Once all the values have been read on all the input channels, in any order, then we can print the values to the console window. This is achieved by the use of a closure that iterates over `each` of the elements in `readerList` {36}. It is assumed that any object printed by this process will have the method `toString()` defined. The variable `pr` is assigned, in turn, to each list element from which we extract the `value` field that can then be printed. If the value of `delay` is greater than zero then the `sleep` method is called on `timer`, which causes this process to stop execution, idle, for at least `delay` milliseconds {38}.

We can now use this process to print out all the intermediate values in the process network shown in Figure 3-10. This is simply achieved by inserting `GPCopy` processes into each connecting channel and sending one output to the next process and the other into the `GParPrint` process as shown in Figure 3-11. Arrays of channels are used to make naming easier as shown in Listing 3-16. The channels `connect` form links between the processes as a long chain or pipeline. The channels `outChans` provide the connection between the intermediate `GPCopy` processes and the final process to the `GParPrint` process.

The output from `GNumbers` is sent via `connect[0]` to the first instance of `GPCopy` which outputs the value in parallel to `connect[1]` and `outChans[0]`. Channel `connect[1]` then forms the input to `GIntegrate`, the output from which is communicated on `connect[2]` to the second instance of `GPCopy`. Channel `connect[3]` then sends the data stream to an instance of `GPairs`, the output of which is sent via `connect[4]` to an instance of `GPrefix`, which then finally sends the stream to `outChans[2]`. The `GPrefix` process has been inserted so that the tabular output is formatted correctly with a first line of zeros. Recall that `GPairs` consumes the first pair of numbers and only outputs a single number; hence we need to insert another number, 0, to form the tabular output correctly.

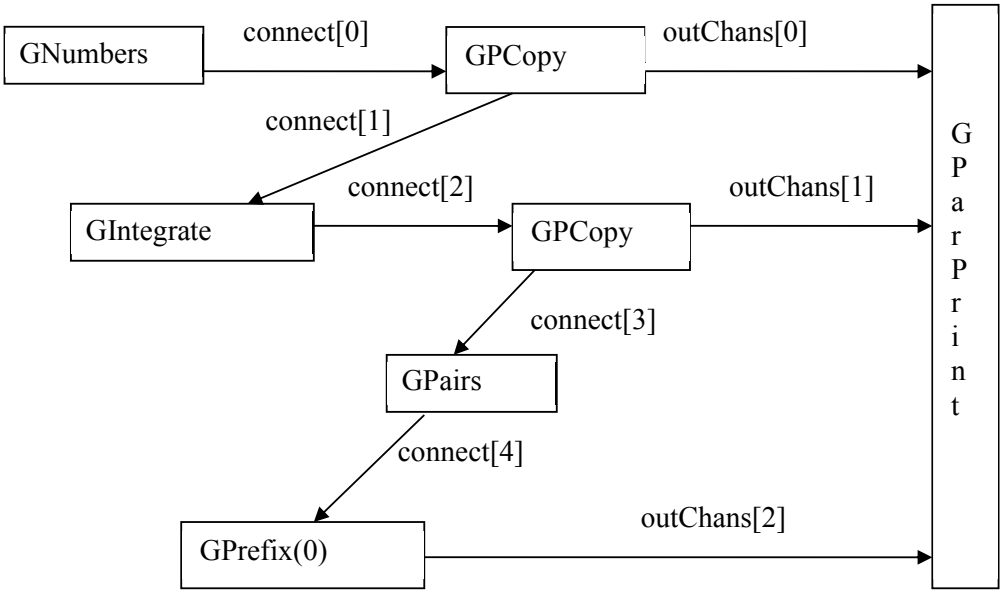


Figure 3-11 The Squares Network with Additional Printing

The arrays of channels are defined using a channel array constructor as shown in {10–11}, Listing 3-16. The lists of inputs to GParPrint are created by means of the constructor for `ChannelInputList`, which takes a parameter of an array of channels and returns a list of channel input ends {13}. The list of `Strings` that make the `titles` of the columns is then defined {15}. The list of processes as shown in Figure 3-11 is then created connecting all the processes together {17–33}. Finally, the list of processes is invoked {35} and produces the output shown in Output 3-4.

```
10 def connect = Channel.one2oneArray(5)
11 def outChans = Channel.one2oneArray(3)
12
13 def printList = new ChannelInputList ( outChans )
14
15 def titles = [ "n", "int", "sqr" ]
16
17 def testList = [ new GNumbers ( outChannel: connect[0].out() ),
18                  new GPCopy ( inChannel: connect[0].in(),
19                               outChannel0: connect[1].out(),
20                               outChannel1: outChans[0].out() ),
21                  new GIntegrate ( inChannel: connect[1].in(),
22                                   outChannel: connect[2].out() ),
23                  new GPCopy ( inChannel: connect[2].in(),
24                               outChannel0: connect[3].out(),
25                               outChannel1: outChans[1].out() ),
26                  new GPairs ( inChannel: connect[3].in(),
27                               outChannel: connect[4].out() ),
28                  new GPrefix ( prefixValue: 0,
29                               inChannel: connect[4].in(),
30                               outChannel: outChans[2].out() ),
31                  new GParPrint ( inChannels: printList,
32                               headings: titles )
33                  ]
34
35 new PAR ( testList ).run()
```

Listing 3-16 Script to Invoke the Process Network Shown in Figure 3-11

```
n     int   sqr
0     0     0
1     1     1
2     3     4
3     6     9
4     10    16
5     15    25
6     21    36
7     28    49
8     36    64
9     45    81
10    55    100
11    66    121
12    78    144
13    91    169
14    105   196
15    120   225
16    136   256
```

Output 3-4 Table of Numbers Showing Intermediate Stages in the Calculation of Squares

Consideration of the output shows that the numbers do appear in sequence in the column headed "n". The column headed "int" does contain the running sum or integration of the numbers. If we ignore the zero appearing in the first row of the column of squares headed "sqr", which was generated by the `GPrefix` process, then we see that there is indeed a list of the squares of the numbers in the first column.

3.10 Summary

We are now able to see why we can assert that this style of parallel processing has a compositional semantics. We know that each process is correct in its own right. By using them together, in a composition, we can go from a statement of what is required; generate a 0, generate a 1, then add the sequence up in pairs to a network that directly implements the requirement. We have also reused previously defined processes. This reuse and compositional capability means the system designer simply has to understand the operation of each of the processes in terms of the use of a process' input and output channels, so they can be correctly connected to each other. The designer does not have to refer to other object definitions to understand the behaviour of a process. It is for this reason that the types of channels have been specified for class properties even though Groovy does not specifically require this to be done. In this simple case we have not specified the nature of the object that is communicated over the channels, as they are all of type `Integer`. In more complex systems the objects to be communicated should be documented as well.

Of more importance, is we have reused a number of processes, in relatively simple networks, to create a number of interesting results. We have also learnt that it is better to reuse existing processes wherever possible, rather than writing new processes, even if this means that we have to write another process. Parallel processing is not just a means of executing systems over a number of processors it also allows us to design systems more easily by composing existing processes into larger systems.

3.11 Exercises

Exercise 3-1

Write a process that undoes the effect of `GIntegrate`. This can be achieved in two ways, first, by writing a `Minus` process that subtracts pairs of numbers read in parallel similar to `GPlus` or by implementing a `Negator` process and inserting it before a `GPlus` process. Implement both approaches and test them. Which is the more pleasing solution? Why?

Exercise 3-2

Write a sequential version of `GPCopy`, called `GSCopy` that has the same properties as `GPCopy`. Make a copy of Listing 3-13 replacing `GPCopy` by your `GSCopy` and call it `GSPairsA`. Create another version, called `GSPairsB` in which the output channels `outChannel0` and `outChannel1` are assigned to the other actual channel, that is `a.out()` is assigned to `outChannel1` and `b.out()` is assigned to `outChannel1`. Take Listing 3-14 as the basis and replace `GPairs` by `GSPairsA` or `GSPairsB` and determine the effect of the change. Why does this happen? The accompanying web site contains the basis for this exercise apart from the body of `GSCopy`. Hint: read Section 3.7.2 that describes the operation of `GTail`.

Exercise 3-3

Why was it considered easier to build `GParPrint` as a new process rather than using multiple instances of `GPrint` to output the table of results?

Exercise 3-4

A `ChannelInputList` has a `read()` method that inputs from each channel in the channel list in parallel and returns a list, the same size as the `ChannelInputList` containing the object that has been read from each channel in the `ChannelInputList`. Modify the coding of Listing 3-15 to make use of this capability.

4 Parallel Processes: Non Deterministic Input

The concept of the non-determinism is defined

- the behaviour of an alternative is described together with the ALT helper class
- a simple example is used, based on processes from the previous chapter
- the concepts of guards and guarded commands are explained
- processes that enable window based user input and output are used

In many systems there is a requirement to provide feedback from a downstream stage of the process to an upstream one. The upstream process has no idea when such a piece of feedback information is going to arrive and thus has to be able to accept it at any time. The behaviour of such a process is said to be non-deterministic because the arrival of the information cannot be determined when the process is defined. We know that such feedback can arrive but not when. Similarly, a process network may be subject to external interventions that change the operation of the system. It is known that these interventions will occur but not when.

For this purpose Alternative provides the program structuring mechanism. In its simplest form the Alternative manages a number of input channels. On executing an Alternative the state of all the input channels is determined.

If none of the channels are ready the Alternative waits until one is ready, reads the input and then obeys the code body associated with that input.

If one input is ready then that channel is read and its associated code is obeyed.

If more than one channel is ready then one is chosen according to some selection criterion and the channel is read and its associated code body obeyed. Typically, an Alternative is incorporated into a looping structure so that the input channels can be repeatedly accessed.

As a first example we shall take the process that generates a sequence of integers, `GNumbers`, previously described in Section 3.4. We shall modify it to accept an input which resets the sequence to a number input by the user at any time chosen by the user.

4.1 Reset Numbers

The structure of the revised numbers process is shown in Figure 4-1.

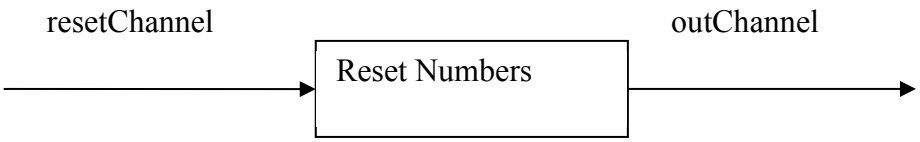


Figure 4-1 The Reset Numbers Process Structure

This however does not indicate the changes required to the internal operation of the ResetNumbers process, which is shown in Figure 4-2.

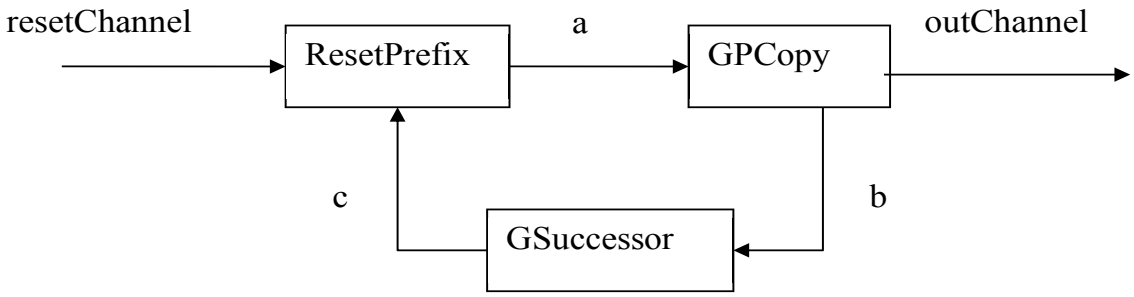


Figure 4-2 Internal Structure of ResetNumbers Process

Instead of using the `GPrefix` process of `GNumbers` (see Figure 3-4) we use a new process `ResetPrefix`. Figure 4-2 exposes the fact that the `ResetPrefix` process contains two input channels over which it can alternate. The coding of the `ResetPrefix` process is shown in Listing 4-1.

```
10 class ResetPrefix implements CSProcess {
11
12   def int prefixValue = 0
13   def ChannelOutput outChannel
14   def ChannelInput inChannel
15   def ChannelInput resetChannel
16
17   void run () {
18     def guards = [ resetChannel, inChannel ]
19     def alt = new ALT ( guards )
20     outChannel.write(prefixValue)
21     while (true) {
22       def index = alt.priSelect()
23       if (index == 0 ) { // resetChannel input
24         def resetValue = resetChannel.read()
25         inChannel.read()
26         outChannel.write(resetValue)
```

```
27          }
28          else { //inChannel input
29            outChannel.write(inChannel.read())
30          }
31        }
32      }
33 }
```

Listing 4-1 ResetPrefix Coding

The properties of the process comprise {12–15}; the initial `prefixValue` from which the first sequence will start. The channels it uses to communicate, namely, `inChannel`, `outChannel` and `resetChannel`. The latter receives the value to which the sequence of integers is to be reset.

An alternative comprises a number of guards and associated guarded commands. The construction of an alternative is assisted by the `ALT` helper class. In this case a guard is simply an input channel and the guarded command is the code body that is associated with that channel input. There are two `guards`{18}, formed as a `List`; the `resetChannel` and `inChannel`. The latter is used during the normal operation of the process network. The order in which these guards are specified is important because we wish to give priority to the `resetChannel`. The alternative `alt` is defined {19} by means of the `ALT` helper class, which takes a `List` of guards.

The fundamental operation of the process remains the same as `GPrefix`, in that the value of `prefixValue` is output {20} after which the process repeatedly {21–31} inputs a value and then outputs the same value on its `outChannel`. In this case the input comes from either the `resetChannel` or the `inChannel`. The method `priSelect()` applied to `alt` {22} returns the `index` of the channel that was selected from the alternative using the criterion that the channel selected has the lowest `List` index if more than one guard is ready. Thus an `index` value of 0 implies that the `resetChannel` is ready to be read from and 1 implies that `resetChannel` was not ready and `inChannel` was ready. Hence we can construct a simple `if` statement {23-30} to discriminate these cases for each of the guards. In more complex alternatives, with more guards, we would use a `switch` statement. The first operation of any guarded command sequence must be to read from the channel that was selected by the alternative.

The guarded command for the `resetChannel` reads from the channel {24} and assigns its value to `resetValue`. The value currently circulating round the network needs to be read from `inChannel` {25} and ignored after which `resetValue` can be written to the `outChannel` {26}. The guarded command for input from `inChannel` is identical to the original version of `GPrefix` in that a value is read from `inChannel` into `inputValue` {29} and then written to the `outChannel` {29}.

Listing 4-2 shows the coding of the `ResetNumbers` process, which can be seen to be a direct implementation of Figure 4-2.

```
10 class ResetNumbers implements CSProcess {
11
12   def ChannelOutput outChannel
13   def ChannelInput resetChannel
14   def int initialValue = 0
15
16   void run() {
17
18     def a = Channel.one2one ()
19     def b = Channel.one2one()
20     def c = Channel.one2one()
21
22     def testList = [ new ResetPrefix ( prefixValue: initialValue,
23                                        outChannel: a.out(),
24                                        inChannel: c.in(),
25                                        resetChannel: resetChannel ),
26                      new GPCopy ( inChannel: a.in(),
27                                   outChannel0: outChannel,
28                                   outChannel1: b.out() ),
29                      new GSuccessor ( inChannel: b.in(),
30                                       outChannel: c.out())
31                      ]
32     new PAR ( testList ).run()
33   }
34 }
```

Listing 4-2 Definition of The ResetNumbers Process

4.2 Exercising ResetNumbers

In order to exercise `ResetNumbers` a process is required that can send values to its `resetChannel`. This is most simply achieved by running two processes in parallel, each with their own user interface so the interaction between the processes can be observed. The structure of this process network is shown in Figure 4-3. The user interface is implemented using a GConsole process, see accompanying documentation.

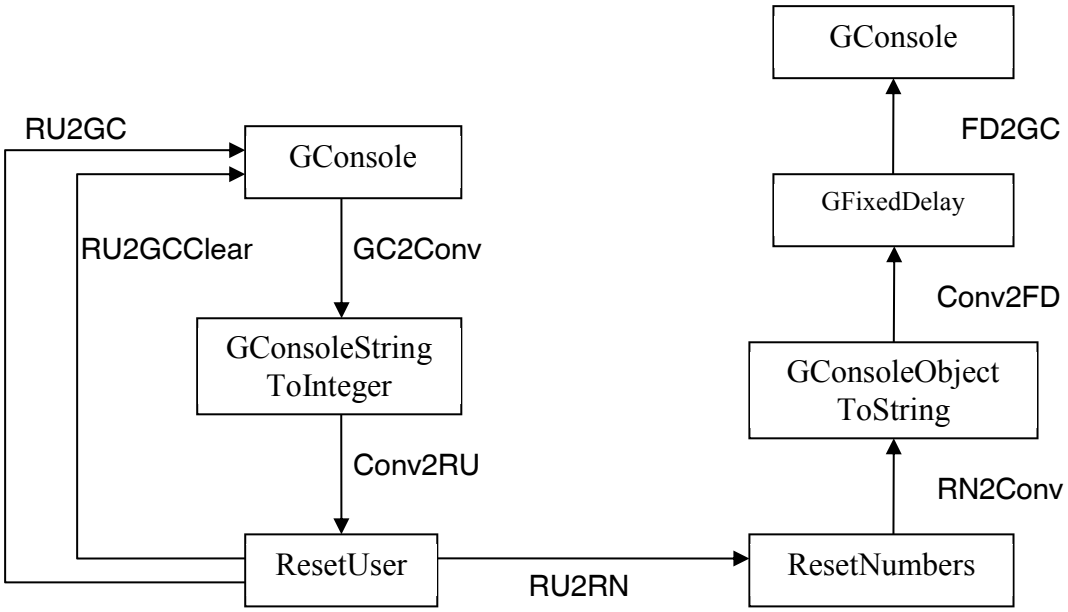


Figure 4-3 Exercising the ResetNumbers Process

The `ResetUser` process receives inputs from its `GConsole` user interface process through a `GConsoleStringToInteger` process, which coverts an input string typed into the input area of the user interface into an `Integer`. The channel `RU2GC` is used to output messages to the user interface's output area. The channel `RU2GCClear` is used to clear the user interface's input area between inputs. On receiving an input, the `ResetUser` process outputs this value to the `ResetNumbers` process using the channel `RU2RN`. This value is input on the `ResetNumber`'s `resetChannel` (see Figure 4-2 and Listing 4-2), which is then communicated to the `resetChannel` of the `resetPrefix` process (see Listing 4-1 {15, 24}) contained within the `ResetNumbers` process.

The process `ResetNumbers` outputs a sequence of numbers, the content of which changes as reset values are read, to its `RN2Conv` channel. The process `GConsoleObjectToString` process converts any `object` to a `String` using the object's `toString()` method. The `String` representation of the values passes through a process called `GFixedDelay` which introduces a delay into the output stream sent to the `GConsole` process so that it is easier to read from the user interface's output area.

Listing 4-3 presents the coding of the `ResetUser` process. An initial message is written to the user interface {18} after which values are repeatedly {19–23} read from the `GConsoleStringToInteger` process as an `Integer` value `v` {20}, after which the input area of the user interface is cleared {21}. The value to be sent to the reset channel is then written to the channel `resetValue` {22}.

```
10 class ResetUser implements CSProcess {
11
12   def ChannelOutput resetValue
13   def ChannelOutput toConsole
14   def ChannelInput fromConverter
15   def ChannelOutput toClearOutput
16
17   void run() {
18          toConsole.write( "Please input reset values\n" )
19          while (true) {
20           def v = fromConverter.read()
21           toClearOutput.write("\n")
22           resetValue.write(v)
23          }
24   }
25 }
```

Listing 4-3 The ResetUser Process

Listing 4-4 shows the implementation of the process network shown in Figure 4-3.

```
10 def RU2RN = Channel.one2one()
11
12 def RN2Conv = Channel.one2one()
13 def Conv2FD = Channel.one2one()
14 def FD2GC = Channel.one2one()
15
16 def RNprocList = [ new ResetNumbers ( resetChannel: RU2RN.in(),
17                                        initialValue: 1000,
18                                        outChannel: RN2Conv.out() ),
19                   new GObjectToConsoleString ( inChannel: RN2Conv.in(),
20                                                outChannel:Conv2FD.out() ),
21                   new GFixedDelay ( delay: 200,
22                                     inChannel: Conv2FD.in(),
23                                     outChannel: FD2GC.out() ),
24                   new GConsole ( toConsole: FD2GC.in(),
25                                     frameLabel: "Reset Numbers Console" )
26             ]
27
28 def RU2GC = Channel.one2one()
29 def GC2Conv = Channel.one2one()
30 def Conv2RU = Channel.one2one()
31 def RU2GCClear= Channel.one2one()
32
33 def RUprocList = [ new ResetUser ( resetValue: RU2RN.out(),
34                                     toConsole: RU2GC.out(),
35                                     fromConverter: Conv2RU.in(),
36                                     toClearOutput: RU2GCClear.out()),
37                      new GConsoleStringToInteger ( inChannel: GC2Conv.in(),
38                                     outChannel: Conv2RU.out()),
39                       new GConsole ( toConsole: RU2GC.in(),
40                                     fromConsole: GC2Conv.out(),
41                                     clearInputArea: RU2GCClear.in(),
42                                     frameLabel: "Reset Value Generator" )
43             ]
44 new PAR (RNprocList + RUprocList).run()
```

Listing 4-4 The Script That Exercises the ResetNumbers Process

Initially, the channel `RU2RN` is defined {10}. After which the two parts of the network are defined separately. First, the `ResetNumbers` network is defined {12–26}. The channels `RN2Conv`, `Conv2FD` and `FD2GC` are defined {12–14}. The list `RNprocList` contains instances of each of the processes shown in Figure 4-3 used to implement the `ResetNumbers` process and its interface components. By inspection, it can be seen the connections shown in Figure 4-3 are implemented by the properties passed to each of the processes in the list {16–26}. The processes `GObjectToConsoleString`, `GFixedDelay` and `GConsole` are described in more detail in the accompanying documentation. The timing of the stream of output numbers is governed by the `delay` property of the `GFixedDelay` process {21–23}.

The second part of Listing 4-4 {28–43} shows the network used to implement the `RUprocList` that implement the `ResetUser` part of the system and its interface components. The channels `RU2GC`, `GC2Conv`, `Conv2RU` and `RU2GCClear` implement the channels shown in Figure 4-3 {28–31}. Finally, the two process lists `RNprocList` and `RUprocList` are concatenated using the overloaded + operator {44} and the network is executed. Each of the parts of the network has their own `GConsole` process. The `frameLabel` property of this process is used to write a title on each of the user interface windows {25, 42} respectively.

When the processes are invoked it can be observed that as reset values are typed into the `Reset Value Generator` console, the values in the `Reset Numbers Console` continue for a short time with the original sequence and then produce a sequence starting with the recently typed reset value.

4.3 Summary

This chapter has introduced the concept of the alternative and shown how it can be used to choose amongst a number of input channels. In the next chapter we shall show how the guards can be extended to include timer alarms in a more realistic example derived from machine tool control.

4.4 Exercises

Exercise 4-1

What happens if line {25} of `ResetPrefix` Listing 4-1 is commented out? Why?

Explore what happens if you try to send several reset values hence, explain what happens and provide a reason for this.

Exercise 4-2

Construct a different formulation of `ResetNumbers` that connects the reset channel to the `GSuccessor` process instead of `GPrefix`. You will have to write a `ResetSuccssor` process. Does it overcome the problem identified in Exercise 1? If not, why not?

5 Extending the Alternative: A Scaling Device and Queues

This chapter demonstrates further capabilities of the alternative by:

- setting timer alarms and using them as alternative guards
- showing that alternatives can be nested
- incorporating pre-conditions into alternatives

Many machines used in automated processes have some means of monitoring their operation, for example, by calculating running averages of specific values and ensuring they stay within a specified range. If they go out of range then the machine recalibrates itself. In this chapter we shall build a model of such a device, but without having to interface to a real machine! This happens for example in medical laboratory equipment where a running check is kept of the range of values for each test that has been produced. Over a given period it is known that the mean value will lie within known bounds. If the machine is out with those bounds then it enters an automatic recalibration process.

5.1 The Scaling Device Definition

The scaling device reads (Belapurkar, 2013) incoming integers that arrive every second. The device then multiplies the incoming value by its current scaling factor, which it then outputs, together with the original value. The scaling factor is doubled at a regular interval, of say, 5 seconds. In addition, there is a controlling function that suspends the operation of the scaling device again at regular intervals, of say, 11 seconds to simulate the testing of its operation. When it is suspended the scaling device outputs its current scaling factor to the controller. At some time later, the controller, having computed another scaling factor, will inject the new scaling factor into the controller, which resumes its normal mode of operation. While the scaling device is suspended by the controller it outputs all input values unscaled.

The structure of the system, showing the channels that will be used for the communications specified above is shown in Figure 5-1.

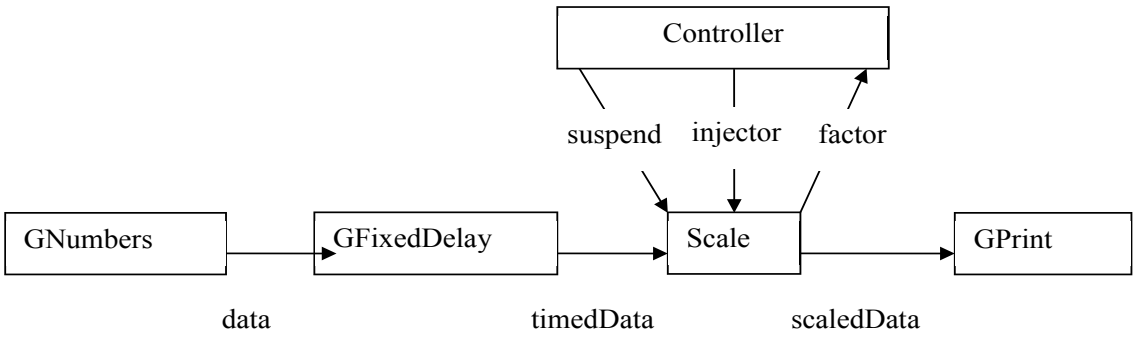


Figure 5-1 Structure of the Scaling Device

The processes `GNumbers`, `GFixedDelay` and `GPrint` are available in the package `groovyPlugAndPlay`. Thus the discussion revolves around the structure of the remaining two processes.

5.1.1 The Controller Process

The code that implements the `Controller` process is shown in Listing 5-1.

```
10 class Controller implements CSProcess {
11
12   def long testInterval = 11000
13   def long computeInterval = 2000
14   def int addition = 1
15
16   def ChannelInput factor
17   def ChannelOutput suspend
18   def ChannelOutput injector
19
```

```
20    void run() {
21      def currentFactor = 0
22      def timer = new CSTimer()
23      def timeout = timer.read()
24
25      while (true) {
26        timeout = timeout + testInterval
27        timer.after ( timeout )
28        suspend.write (0)
29        currentFactor = factor.read()
30        currentFactor = currentFactor + addition
31        timer.sleep(computeInterval)
32        injector.write ( currentFactor )
33      }
34    }
35 }
```

Listing 5-1 Code of the Controller Process

From Figure 5-1 we can see that `Controller` has three channel properties {16–18}. In addition, it has two timeout values, one `testInterval` {12} determines the period between successive tests of the scaling device, which has a default value of 11 seconds. The other, `computeInterval {13}` is used to simulate the time it takes to compute the revised scaling factor, which has a default value of 2 seconds. All times are expressed in milliseconds.

The JCSP class `CSTimer` provides a means of manipulating time in a consistent and coherent manner. An instance of `CSTimer`, called `timer` is defined {22}. The `timer` can be read at any instant and the current `long` value of the system clock in milliseconds is returned, which also justifies the type `long` for the interval properties defined previously. The value of `timeout` is set to the current time {23}. The device operates as a never ending loop {25–33}, which for most automated tools is reasonable.

Within the loop the timeout is incremented by the `testInterval` {26}, which must be some time in the future. The `after` operation on `timer` causes the process to be suspended until the value of the current time is after the indicated alarm time. While a process is suspended in this manner it will consume no processing resource. Once the `testInterval` has elapsed, the `Controller` writes a signal to the `Scale` process to suspend its operation {28}. The value communicated does not matter, so the value 0 is perfectly adequate. The `Controller` then reads the current scaling factor from the `Scale` process into `currentFactor` using the channel `factor` {29}. The value of `currentFactor` is then modified {30} by the value contained in the property `addition` {14} (default value 1), to simulate a change in the scaling factor. The time to undertake this recalculation is then simulated by suspending the process for the `computeInterval` by calling the `sleep` method on the `timer` {31}. The `sleep` method deschedules the process for the specified sleeping time.

The process consumes no processor resource while it is sleeping. In this case the effect of `after` and `sleep` are the same, achieved in a different manner. In some situations, the `after` method will be the more appropriate because it provides relative time. The `sleep` method provides an absolute value. Once the process has been rescheduled, it writes {32} the newly computed `currentFactor` on the `injector` channel to the `Scale` process.

5.1.2 The Scale Process

The structure of the `Scale` process is shown in Listings 5-2 and 5-3. The operation of the `Scale` process can be partitioned into two distinct parts; when it is operating in the normal mode and when it is suspended. In the normal mode it accepts inputs from the channels `timedData` and `suspend`, see Figure 5-1. It will also respond to timer alarms indicating that the scaling factor should be doubled. In the suspended mode it will only respond to inputs from the channels `timedData` and `injector`. To reflect these situations a set of guards will be needed for each mode. Furthermore, the suspended set will only be considered when the process has moved from the normal mode into the suspended mode.

```
10 class Scale implements CSProcess {
11
12   def int scaling = 2
13   def int multiplier = 2
14
15   def ChannelOutput outChannel
16   def ChannelOutput factor
17   def ChannelInput inChannel
18   def ChannelInput suspend
19   def ChannelInput injector
20
21   void run () {
22     def SECOND = 1000
23     def DOUBLE_INTERVAL = 5 * SECOND
24     def NORMAL_SUSPEND  = 0
25     def NORMAL_TIMER    = 1
26     def NORMAL_IN       = 2
27     def SUSPENDED_INJECT = 0
28     def SUSPENDED_IN     = 1
29     def timer = new CSTimer()
30     def normalAlt = new ALT ( [ suspend, timer, inChannel ] )
31     def suspendedAlt = new ALT ( [ injector, inChannel ] )
32     def timeout = timer.read() + DOUBLE_INTERVAL
33     timer.setAlarm ( timeout )
```

Listing 5-2 The Properties and Initialisation of the Scale Process

The channel properties are defined {15–19}, together with the initial `scaling` value {12} and the `multiplier` that will be applied to the scaling factor {13}. The `inChannel` property {17} is connected to `timedData` of Figure 5-1 and `outChannel` to `scaledData` {15}. Within the `run()` method a number of constants are defined; `DOUBLE_INTERVAL` {23} specifies the number of milliseconds between the doubling of the `scaling` factor. The remainder are constants {24–28} used to identify which `case` is to be considered when the `switch` statements associated with the alternatives are processed. A `timer` is defined {29}, followed by the two different alternatives {30, 31}. Both of the alternatives will be accessed using a `priSelect` method and thus the ordering of the guards in the alternatives is important and should always start with the highest priority going to the first in sequence. The alternative `normalAlt` applies when the device is not in a `suspended` state. The highest priority guard is that associated with the `suspend` channel. The next highest will result from a `timer` alarm and the lowest is the input of some data on the `inChannel`. In the `suspended` state the `suspendedAlt` will apply and this is just an alternation over the `injector` and `inChannel` channels because `timer` alarms are ignored. At {32} the `timeout` for the first doubling of the scaling factor is defined by reading the `timer` and adding the doubling interval. An alarm on the `timer` is made by calling the method `setAlarm` {33} with the required time, which must be some time in the future. This means that `normalAlt` {30} will be enabled on the `timer` alternative once the value of the `timer` has increased beyond `timeout`. A timer contained within an alternative guard that is disabled, consumes no processor resource, until the alarm is enabled.

```
34      while (true) {
35        switch ( normalAlt.priSelect() ) {
36
37          case NORMAL_SUSPEND :
38            suspend.read()
39            factor.write(scaling)
40            def suspended = true
41            println "Suspended"
42            while ( suspended ) {
43
44              switch ( suspendedAlt.priSelect() ) {
45
46                case SUSPENDED_INJECT:
47                  scaling = injector.read()
48                  println "Injected scaling is $scaling"
49                  suspended = false
50                  timeout = timer.read() + DOUBLE_INTERVAL
51                  timer.setAlarm ( timeout )
52                  break
53
54                case SUSPENDED_IN:
55                  def inValue = inChannel.read()
56                  def result = new ScaledData()
57                  result.original = inValue
58                  result.scaled = inValue
59                  outChannel.write ( result )
60                  break
61              } // end-switch
62            } //end-while
63            break
64
65          case NORMAL_TIMER:
66            timeout = timer.read() + DOUBLE_INTERVAL
67            timer.setAlarm ( timeout )
68            scaling = scaling * multiplier
69            println "Normal Timer: new scaling is $scaling"
70            break
71
72          case NORMAL_IN:
73            def inValue = inChannel.read()
74            def result = new ScaledData()
75            result.original = inValue
76            result.scaled = inValue * scaling
77            outChannel.write ( result )
78            break
79
```

```
80        } //end-switch
81      } //end-while
82    } //end-run
83 } // end Scale
```

Listing 5-3 The Scale Process Main Loop

The main loop of the scaling device, Listing 5-3, comprises {34–81} and is created by means of a never ending `while` loop {34}. At the start of the main loop the device is presumed to be in the normal state and thus we `switch` on the `normalAlt` {35}. If none of the guards is ready the process waits until one becomes enabled. Each time an alternative is executed the guards are evaluated to determine which are enabled and then a selection is made from the ready ones according to the type of `select` operation undertaken. In this case a `priSelect()` is deemed more appropriate.

If the enabled alternative results from an input on the `suspend` channel then the `case` `NORMAL_SUSPEND` will be obeyed {37}. First, the channel `suspend` must be `read {38}`, the value of which can be ignored because this is just a signal to enter the suspended state. Recall that the `Controller` process wrote a nominal value of 0 (Listing 5-1 {28}). The `Scale` process then writes its current `scaling` factor to the `factor` channel {39}. The property `suspended` is defined and set `true` {40}. A message is printed {41} and then the loop associated with the `suspended` state is entered {42}. In this state the process `switch`es on `suspendedAlt` {31}, which has two alternatives.

If the enabled alternative is an input on the `injector` channel the case `SUSPENDED_INJECT` is obeyed {46}. The new value of `scaling` is `read` from the `injector` channel {47} and a message displaying the new factor printed {48}. The value of `suspended` is now reset {49} to `false`, which will cause the controlling while loop {42} to terminate. Because the `injector` input is also taken as an indication that normal operation can resume, the `timer` alarm can be reset {50–51}.

In the `suspended` state, the only other alternative that can occur, results from input on the `inChannel`, this causes the `SUSPENED_IN` case to be obeyed {54}. The channel `inChannel` is `read` into `inValue` {55}. A variable `result` of type `ScaledData` is defined {56}, see Listing 5-4. The device in the `suspended` state does not apply the `scaling` to any incoming data and so both the `original` and `scaled` values of `result` are set to `inValue` {57–58}. The `result` object is then written to `outChannel` {59}.

The remaining cases relate to the operation of the device in the normal state. If a `timer` alarm occurs the code associated with the `NORMAL_TIMER` case is obeyed {65}. The `timer`'s `timeout` alarm is reset for the next doubling period {66–67}. The scaling is multiplied by multiplier, which is 2 for doubling {68} as required by the device specification and an appropriate message printed {69}. The final case deals with inputs from `inChannel` {72}. The value is read from `inChannel` into `inValue` {73} and placed in the `original` property {75} of a new `result` object {74}. A scaled value is placed in the `scaled` property of a new `result` object {76}, which is then written to `outChannel` {77}.

5.1.3 The ScaledData Object

The `ScaledData` object is used to pass a pair of values from the `Scale` process to the `GPrint` process see Figure 5-1. Its structure is shown in Listing 6-3.

```
10 class ScaledData implements Serializable {
11
12  def int original
13  def int scaled
14
15  def String toString () {
16      def s = " " + original + "\t\t" + scaled
17      return s
18    }
19 }
```

Listing 5-4 The ScaledData Object

The properties of the object; `original` and `scaled` are defined {12, 13} and then a `toString`() method is defined {15–18} that is used when the object is printed.

More importantly, this is the first instance of user defined objects being communicated between processes. The first aspect to notice is there are no public data manipulation methods, other than implicit getters and setters that are created by the Groovy environment automatically, because in the parallel environment we encapsulate the data so that it is processed only within processes. It is not possible for one process to access another process object's properties to modify its state by calling public methods.

Concurrent processes pass object references over channels and thus a sending process has to guarantee that once it has written an object to a channel it does not modify that object in any way. This is most easily achieved by defining a new object instance for each write operation, see Listing 5-3 {56, 74}. In some cases, it may be necessary, for memory management reasons; to reuse an object and to ensure that a written object is not overwritten a deep copy is taken. An interface `JCSPCopy` which contains a single method `copy()` is provided in the `org.jcsp.groovy` package to facilitate this requirement. The programmer has to write the code to achieve the deep copy of the object. This can then be applied recursively to any nested objects.

If an object is to be passed between networked processes then a copy of the object is passed between the processes and so the object must implement the interface `serializable`. In this case it is not necessary to undertake the method `copy` because a new object instance is created every time. The object implements the `Serializable` interface {10} so that were the object to be communicated over a network, then we know that it will be correctly serialized.

5.1.4 Exercising the Scale Device Network

Listing 5-5 gives the script that implements the process network shown in Figure 5-1.

```
10 def data = Channel.one2one()
11 def timedData = Channel.one2one()
12 def scaledData = Channel.one2one()
13 def oldScale = Channel.one2one()
14 def newScale = Channel.one2one()
15 def pause = Channel.one2one()
16
17 def network = [ new GNumbers ( outChannel: data.out() ),
18                 new GFixedDelay ( delay: 1000,
19                                   inChannel: data.in(),
20                                   outChannel: timedData.out() ),
21
22                 new Scale ( inChannel: timedData.in(),
23                                   outChannel: scaledData.out(),
24                                   factor: oldScale.out(),
25                                   suspend: pause.in(),
26                                   injector: newScale.in(),
27                                   multiplier: 2,
28                                   scaling: 2 ),
29
```

```
30                new Controller ( testInterval: 11000,
31                                 computeInterval: 3000,
32                                 addition: -1,
33                                 factor: oldScale.in(),
34                                 suspend: pause.out(),
35                                 injector: newScale.out() ),
36
37                new GPrint ( inChannel: scaledData.in(),
38                                 heading: "Original Scaled",
39                                 delay: 0)
40                ]
41
42 new PAR ( network ).run()
```

Listing 5-5 Script to Exercise the Scale Device

All the output appears in the Eclipse console window with the messages from the `Scale` process intermingled with those from the output of the original and scaled data which appear in `GPrint`. The `delay` property {39} of `GPrint` is set to 0 so that any output is produced immediately. There is sufficient delay, 1 second, within the system caused by the `GFixedDelay` process {18} to observe the process interactions. In this execution of the script some of the default values in the `Controller` process have been replaced by other values {30-35}.

5.2 Managing A Circular Queue Using Alternative Pre-conditions

A *queue* is a common data structure used in many applications. A number of cases have to be considered as follows.

- data can only be put into the queue if there is space in the queue
- data can only be taken from the queue if the queue is not empty

In a sequential implementation these states have to be tested before the queue can be manipulated and dealing with the situations where either a put or get to or from the queue cannot be undertaken can be problematic. A parallel implementation is much easier to design and specify because we can use an alternative with pre-conditions to ensure that operations only take place when it is safe. Figure 5-2 shows the basic structure that will be used to explain the operation of a queue.

The `QProducer` process puts a sequence of integers into the `Queue` process, where they are stored in a wrap-around List implementing a circular queue. The `QConsumer` process attempts to `get` data from the Queue, which, if there is data available, is received by the process.

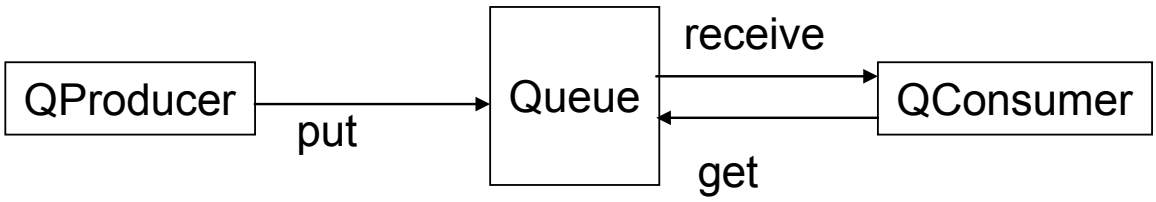


Figure 5-2 The Queue Process Network

5.2.1 QProducer and QConsumer Processes

The source of the `QProducer` process is given in Listing 5-6.

```
10 class QProducer implements CSProcess {
11
12   def ChannelOutput put
13   def int iterations = 100
14   def delay = 0
15
16   void run () {
17     def timer = new CSTimer()
18     println "QProducer has started"
19
20     for ( i in 1 .. iterations ) {
```

```
21        put.write(i)
22        timer.sleep (delay)
23      }
24      put.write(null)
25    }
26 }
```

Listing 5-6 The QProducer Process script

The `timer` {17} is used to create a `delay` {14} between each `write` {21} to the `put` channel. A sequence of integers from 1 up to `iterations` {13} is output on the `put` channel. It should be noted that the `write` on the `put` channel may be delayed {21} if the queue has no available space. Once all the values have been written to the `put` channel a `null` value is also written {24} to indicate that processing has finished. This will be used to terminate the subsequent `Queue` and `QConsumer` processes.

The `QConsumer` process is specified in Listing 5-7. The use of the `timer` and associated `delay` {17, 14} is the same as in `QProducer`. A Boolean `running` is defined {19} and is used to control the main loop of the process. The main loop of the process {21–28} initially writes a signal value of 1 on the `get` channel. The writing of this signal {22} may be delayed if the queue contains no available data. A value is `read` from the `receive` channel {23} into the object `v`. This `read` operation will take place immediately. The value that has been read is printed {24} after which the process is delayed {25}. If the value read is `null` {26} then `running` is set to `false` and the process will terminate at the next iteration of the `while` loop {21}.

```
10 class QConsumer implements CSProcess {
11
12   def ChannelOutput get
13   def ChannelInput receive
14   def long delay = 0
15
16   void run () {
17     def timer = new CSTimer()
18     println "QConsumer has started"
19     def running = true
20
21     while (running) {
22       get.write(1)
23       def v = receive.read()
24       println "QConsumer has read $v"
25       timer.sleep (delay)
26       if ( v == null ) running = false
27     }
28   }
29 }
```

Listing 5-7 The QConsumer Process

5.2.2 The Queue Process

The source for the `Queue` process is shown in Listing 5-8.

The channel properties are defined {12–14} corresponding to Figure 5-2 and the size of the queue is specified in the property `elements` and defaults to 5 {15}. The alternative associated with the `Queue` process is defined as `qAlt`, which has guards comprising the `put` and `get` channels {18}. A `Boolean` array, `preCon`, which has the same number of elements as there are guards in `qAlt,` is defined {19}. Two constants `PUT` and `GET` are defined {20–21} that are used to index the `preCon` array and also to identify the cases in the `switch` statement associated with identifying the selected guard in the alternative.

The array `preCon` is used to record whether or not a new element can be put into the queue storage and similarly whether an element is available. Initially, therefore `preCon[PUT]` is set `true` {22} because there is bound to be space for a new element in the queue data structure because it must be empty. Similarly, `preCon[GET]` is set `false` {23} because there is no data available in the queue. The `List data` {24} provides the storage for the circular queue structure. The variables `count`, `front` and `rear` {25–27} record the state of the queue storage in terms of the number of data values in the queue, the location into which data can be added and removed from the queue respectively. The process is implemented as a loop {30-48}, which is controlled by a `Boolean running` {28} that is set `false` when a `null` value is communicated to the `QConsumer` process {41}.

The variable `index` {31} indicates the alternative guard that has been selected. In order to be selected a guard must have its associated `preCon` element set to `true` and its channel must be enabled to read an input. Note how the pre-condition array is passed as a parameter to the alternative `priSelect` method {31}. A choice is then made depending upon which guard has been selected. Priority is given to inputs from `QProducer` rather than `QConsumer` {18}. It could have been replaced by a call to `Select()` which would have allocated no relative priority between get and put operations.

In the case of `PUT` the value `read` from `put` is placed in `data[front]` {34} and then the values of `count` and `front` are updated appropriately {35–36}. When `GET` is selected, the signal communication on the `get` channel is read and ignored {39}. The value in `data[rear]` is then written to channel `receive` {40}. The value in `data[rear]` is then tested to determine whether the `Queue` process should terminate {41}. The operations have been ordered so that the terminating `null` value is sent to the `QConsumer` process before the `Queue` process terminates. After which, the values of `count` and `rear` are updated {42–43}. At the end of each loop of the queue process, the values stored in the elements of the `preCon` array are updated based upon the relative values of `count` and `elements` {46, 47}.

```
10 class Queue implements CSProcess {
11
12   def ChannelInput put
13   def ChannelInput get
14   def ChannelOutput receive
15   def int elements = 5
16
17   void run() {
18     def qAlt = new ALT ( [ put, get ] )
19     def preCon = new boolean[2]
20     def PUT = 0
21     def GET = 1
22     preCon[PUT] = true
23     preCon[GET] = false
24     def data = []
25     def counter = 0
26     def front = 0
27     def rear = 0
28     def running = true
29
30     while (running) {
31       def index = qAlt.priSelect(preCon)
32       switch (index) {
33         case PUT:
34           data[front] = put.read()
35           front = (front + 1) % elements
36           counter = counter + 1
37           break
38         case GET:
39           get.read()
40           receive.write( data[rear])
41           if (data[rear] == null) running = false
42           rear = (rear + 1) % elements
43           counter = counter - 1
44           break
45       }
46       preCon[PUT] = (counter < elements)
47       preCon[GET] = (counter > 0 )
48     }
49     println "Q finished"
50   }
51 }
```

Listing 5-8 The Queue Process Definition

The benefit of this alternative based formulation is that the pre-condition array modifies the behaviour of its underlying mechanism. Thus if the queue is full then `preCon[PUT]` is `false` and even if there is a communication on the `put` channel it will not be permitted. Similarly, if `preCon[GET]` is `false` then no signal on the `get` channel can be `read`, even if `QConsumer` has tried to write to it, meaning that a get cannot be executed on an empty queue..

5.3 Summary

This chapter has explored the alternative mechanism together with its associated pre-condition Boolean array. It has shown by means of an example based upon a realistic system and one found in many program development applications that alternative has the ability to capture many aspects of real world systems and to provide a flexible means of modelling such systems.

5.4 Exercises

Exercise 51

The accompanying projects contain a script, called RunQueue, in package `ChapterExercises/src/c5` to run the queue network. The delays associated with `QProducer` and `QConsumer` can be modified. By varying the delay times demonstrate that the system works in the manner expected. Correct operation can be determined by the QConsumer process outputting the messages `"QConsumer has read 1"` to `"QConsumer has read 50"` in sequence. What do you conclude from these experiments?

Exercise 52

Reformulate the scaling device so that it uses pre-conditions rather than nested alternatives. Which is the more elegant formulation? Why?

6 Testing Parallel Systems: First Steps

Testing of systems is crucial and this chapter shows how GroovyTestCase can be used to test processes by

- identifying the process that is to be tested which must terminate
- creating support processes that terminate
- constructing the assertions that can be tested, using properties from the support processes

JUnit (JUnit, 2013) testing has become a widely accepted way of testing Java classes and there is a great deal of software support for this approach. In previous chapters, examples and exercises were introduced whereby the user had to ascertain for them self that the systems worked in the expected manner. This was achieved by looking at displayed output. This may be a satisfactory approach for small example systems but is not appropriate for systems that are to be used in an everyday context.

In this chapter the use of JUnit testing is introduced by using examples taken from earlier chapters. This will demonstrate that it is possible to use this approach and give a general architecture for testing parallel systems. The key to JUnit testing is that we test one or more assertions concerning the underlying implementation. In the parallel situation we have to identify a source of inputs that can be compared to the subsequent outputs for the assertion testing.

6.1 Testing Hello World

The testing of the `ProduceHW` and `ConsumeHello` processes (see Chapter 2) demonstrate that from the outset testing has to be considered at the time processes are designed and cannot be retrospectively added. To this end, properties are required that can be accessed once a process has terminated. These properties can then become components in any assertion. In this very simple case the `ProduceHW` process needs no alteration.

6.1.1 Revised ConsumeHelloForTest Process

The revised version of `ConsumeHelloForTest`, see Listing 6-1 requires the addition of a property `message` {13}, which is assigned {18} the values that have been read in from `inChannel` {16, 17}.

```
10 class ConsumeHelloForTest implements CSProcess {
11
12   def ChannelInput inChannel
13   def message
14
```

```
15    void run() {
16      def first = inChannel.read()
17      def second = inChannel.read()
18      message = "${first} ${second}!!!"
19      println message
20    }
21 }
```

Listing 6-1 The Revised Version of ConsumeHW

6.1.2 The HelloWorldTest Script

Listing 6-2 gives the script used to test ProduceHW and ConsumerHW.

The `ProduceHW` process from Chapter 2 is imported {10} into the testcase and thus provides a means of testing that process. The remainder of the coding is that required to build an instance of Groovy`TestCase`. This is the Groovy way of building JUnit tests. This requires the definition of a `void` method {14}, the name of which is prefixed with the word `test` that contains the script necessary to run the processes being tested. The primary requirement is that each of the processes **must** terminate. In many systems this is not feasible as the processes run in a loop that does not terminate. In Chapter 17 we shall see how to test such non-terminating process networks.

```
10 import c02.ProduceHW
11
12 class HelloWorldTest extends GroovyTestCase {
13
14    void testMessage() {
15                def connect = Channel.one2one()
16                def producer = new ProduceHW ( outChannel: connect.out()
)
17                def consumer = new ConsumeHelloForTest ( inChannel:
                  connect.in() )
18
19                def processList = [ producer, consumer ]
20                new PAR (processList).run()
21                def expected = "Hello World!!!"
22                def actual = consumer.message
23                assertTrue(expected == actual)
24    }
25 }
```

Listing 6-2 The HelloWorldTest Script

The crucial elements are that we define each process as an instance {16, 17}. This is required so that we can access the `message` property of `ConsumeHelloForTest` when the system terminates. The processes are then run in parallel {19, 20}. The property `expected` is set to the `String` that should be output {21}. The `actual` value is obtained from the `message` property of `ConsumeHelloForTest` {22}. These are then compared {23} using the `assertTrue` method which produces an indication of whether the test passed.

6.2 Testing the Queue Process

The `Queue` Process discussed in Chapter 5.2 can be tested by sending a known number of test values into the `Queue` from the `QProducer` process and then ensuring that the same values are received by the `QConsumer` process. Listing 6-3 shows the modified `QProducerFortest` process. The only modifications required occur on {15}, where a new `List` property is added called `sequence`, which holds the sequence of produced values and on {23} where each produced value is appended (<<) to `sequence`. The printing of the produced values also has been removed. The `sequence` property is required to ensure we have a value that can be tested once the network of processes being tested has terminated.

```
10 class QProducerForTest implements CSProcess {
11
12   def ChannelOutput put
13   def int iterations = 100
14   def delay = 0
15   def sequence = []
16
```

```
17    void run () {
18      def timer = new CSTimer()
19
20      for ( i in 1 .. iterations ) {
21        put.write(i)
22        timer.sleep (delay)
23        sequence = sequence << i
24      }
25      put.write(null)
26    }
27 }
```

Listing 6-3 The Testable Version of QProducer Called QProducerForTest

Listing 6-4 shows the modified `QConsumerForTest` process, which as in the `QProducerFortest` defines a property that can be externally accessed, called `outSequence` {15}. The processing of the terminating `null` value has been modified {25} so that all the received values are appended to `outSequence` unless it is the `null` value, in which case the value of `running` is set `false`, causing the process to terminate.

```
10 class QConsumerForTest implements CSProcess {
11
12    def ChannelOutput get
13    def ChannelInput receive
14    def long delay = 0
15    def outSequence = []
16
17    void run () {
18      def timer = new CSTimer()
19      def running = true
20
21      while (running) {
22        get.write(1)
23        def v = receive.read()
24        timer.sleep (delay)
25        if ( v != null) outSequence = outSequence << v
26        else running = false
27      }
28    }
29 }
```

Listing 6-4 The Modified QConsumerForTest Process

6.3 The Queue Test Script

Listing 6-5 gives the GroovyTestcase class that causes `Queue` process testing. It takes the same basic structure as that used in the test of the Hello World system. It is important to note that the aim is to test the `Queue` process given in Chapter 5 and not the `QProducer` and `QConsumer` processes. The `Queue` process is imported {10}. The channels required to implement the network defined in Figure 5-2 are specified {14–16}. Instances of each of the processes are then defined {18–22}, so that we can subsequently test the values of the properties `sequence` and `outSequence` of `QProducer` and `QConsumer` respectively. The list of processes is then defined and executed in a `PAR` {23–24}, which must terminate if we are to be able to test the process properties in an assertion {28}.

```
10 import c05.Queue
11 class QueueTest extends GroovyTestCase {
12
13     void testQueue() {
14       def QP2Q = Channel.one2one()
15       def Q2QC = Channel.one2one()
16       def QC2Q = Channel.one2one()
17
18       def qProducer = new QProducerForTest ( put: QP2Q.out(), iterations: 50 )
19       def queue = new Queue ( put: QP2Q.in(), get: QC2Q.in(),
20       receive: Q2QC.out(), elements: 5)
21       def qConsumer = new QConsumerForTest ( get: QC2Q.out(),
22       receive: Q2QC.in() )
23       def testList = [ qProducer, queue, qConsumer ]
24       new PAR ( testList ).run()
25
26       def expected = qProducer.sequence
27       def actual = qConsumer.outSequence
28       assertTrue(expected == actual)
29     }
30 }
```

Listing 6-5 The QueueTest Script

The values of the `expected` and `actual` returned values are obtained from their processes and tested {26–28}. In more complex examples the construction of assertions is likely to be more elaborate depending upon the nature of the data being input and generated.

6.4 Summary

In this chapter we have introduced the concept of testing parallel systems, using the JUnit testing framework within a Groovy environment. The key requirement is that the network of processes must terminate. Further, the processes used to test the operation of the process network under test must contain properties that can be populated with data that can then be tested in one or more assertions. In Chapter 17 we reflect further on the testing of parallel systems and show how we can test systems that are designed not to terminate.

6.5 Exercises

Exercise 6-1

Construct a Test Case for the Three-To-Eight system constructed in the exercise for Chapter 2.

7 Deadlock: An Introduction

Deadlock and livelock are defined and then demonstrated by means of:

- firstly, a trivial example, and
- secondly by means of a more complex but typical situation

Deadlock occurs whenever a network of processes gets into a state where none of the processes is able to continue execution. A similar and related problem is that of livelock, which occurs when part of a process network operates in such a manner as to exclude some of the processes from execution, while others appear to continue execution. A first simple example, based upon the producer – consumer pattern already discussed demonstrates the ease with which a deadlocked system can be created.

7.1 Deadlocking Producer and Consumer

Listing 7-1 gives the coding for a process `BadP`. The process has two channels {12, 13}. Its `run` method initially prints a starting message {16} after which it enters a loop {18}. A message indicating the process is about to write to `outChannel` {19} is printed and then the output takes place {20}. The same actions are then undertaken for a `read` method on `inChannel` {21, 22}. A message indicating that the end of the loop has been reached is printed {23} and the process loops back to {18}.

```
10 class BadP implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14
15   def void run() {
16     println "BadP: Starting"
17
18     while (true) {
19       println "BadP: outputting"
20       outChannel.write(1)
21       println "BadP: inputting"
22       def i = inChannel.read()
23       println "BadP: looping"
24     }
25   }
26 }
```

Listing 7-1 BadP Process Coding

Listing 7-2 gives the coding for an equivalent matching process `BadC`, which has an identical structure to `BadP` except that the messages produced are different.

```
10 class BadC implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14
15   void run() {
16     println "BadC: Starting"
17
18     while (true) {
19       println "BadC: outputting"
20       outChannel.write(1)
21       println "BadC: inputting"
22       def i = inChannel.read()
23       println "BadC: looping"
24     }
25   }
26 }
```

Listing 7-2 BadC Process Coding

When these processes are executed the console displays the messages shown in Output 7-1.

```
BadC: Starting
BadC: outputting
BadP: Starting
BadP: outputting
```

Output 7-1 Messages Resulting from the Parallel Execution of BadP and BadC

It can be seen that both processes start and achieve an output at lines {19} respectively but cannot make further progress. The reason for this can be seen by inspection because both processes are attempting to output to each other at the same time and neither can undertake the corresponding input operation. It is shown diagrammatically in Figure 7-1, in which increasing time is represented down the page.

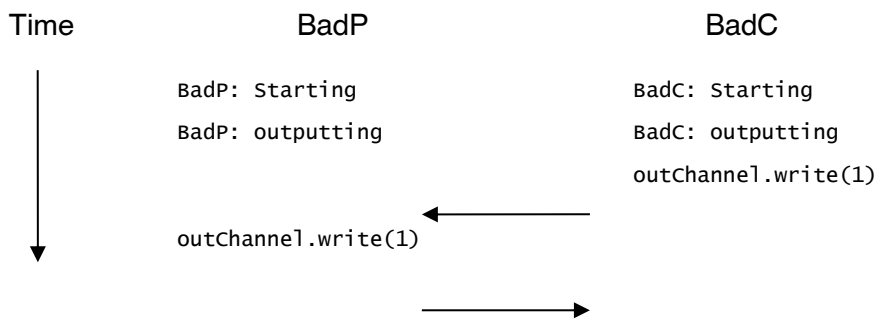


Figure 7-1 Diagrammatic Representation of Deadlocked Process Interactions

In this simple situation the outcome is obvious and easy to see, both processes are trying to output at the same time and thus neither can progress any further because neither can undertake the matching channel read operation. In more complex process networks this is much more difficult to see. Tools are available such as FDR (Formal Systems Europe Ltd, 2013) and Spin (Holzmann G.J., 2013) which can analyse networks of processes for deadlock and livelock but these are limited in the scale of network that can be processed. A different solution is available which can avoid deadlock by engineering design but first we shall investigate a more complex example.

7.2 Multiple Network Servers

A common feature of modern networks is the ability to access many servers from the same workstation. In the background, the network administrator may implement some form of mirror system so that the servers are backed up on each other. When such systems were installed in early network designs there often occurred periods when the network ran very slowly or actually came to halt. The only recourse was to reboot the servers. The problem was this situation was unpredictable.

In this section we shall build a pair of servers that operate in a naïve manner and exhibit this behaviour simply by the order in which data is accessed. The behaviour only requires two clients, which are able to access data from each of the servers but which, crucially access the data through only one server. The system structure is shown in Figure 7-2. Each server has one client and if the client needs to access data that is not available on its own server an automatic access from its server is made to the other server. In order to improve performance of the servers we will allow their clients to initiate another request for data if the previous request has been passed to the other server. Thus requests are interleaved. The connections are shown as double ended arrows to indicate there is a request phase and the subsequent return of a data value corresponding to the request.

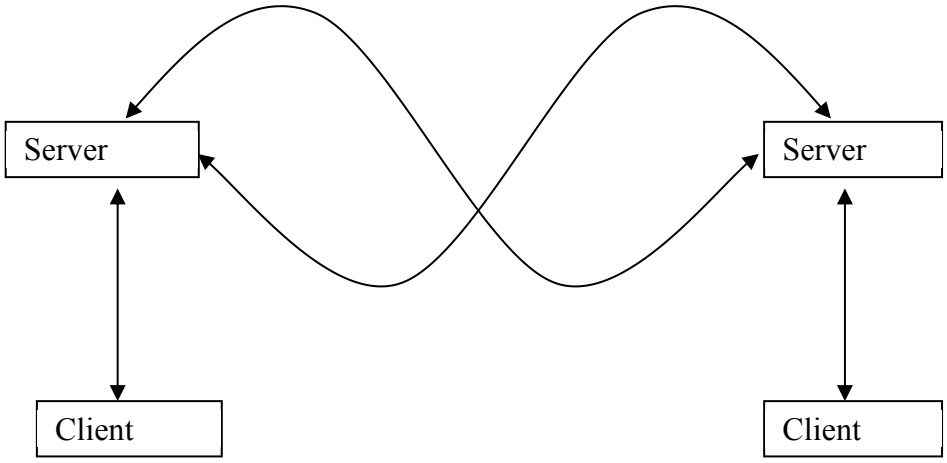


Figure 7-2 Cross-Coupled Clients and Servers System

The servers are implemented as `Map`s comprising 10 elements in each server. The keys for the map entries are distinct. Each client has a list of map entries it wishes to access by key value.

7.2.1 The Client Process

The coding for a Client process is shown in Listing 7-3. The `Client` process has two channel properties `requestChannel` {13}, which is used to send requests to its `Server` and `receiveChannel` {12} used to return results from the `Server`, to the `Client`. The property `selectList` {15} is a `List` initialised to the set of entries in the `Server`'s `Map` that are to be accessed.

```
10 class Client implements CSProcess{
11
12   def ChannelInput receiveChannel
13   def ChannelOutput requestChannel
14   def clientNumber
15   def selectList = [ ]
16
17   void run () {
```

```
18      def iterations = selectList.size
19      println "Client $clientNumber has $iterations values in $selectList"
20
21      for ( i in 0 ..< iterations) {
22        def key = selectList[i]
23        requestChannel.write(key)
24        def v = receiveChannel.read()
25      }
26
27      println "Client $clientNumber has finished"
28    }
29 }
```

Listing 7-3 Client Process Structure

The number of elements in the `selectList` is found {18} and this is used as the range of a `for` loop {21–25}. The client process identity and the list of keys it is going to access are then printed out. Each `key` is found in sequence {22} and this is written to the `requestChannel` {23}. The `Client` then reads the response from the `Server` {24}. At this point the process could print out the returned value but we choose not to. It should be noted that the `Client` may have to wait for the response from its `Server` {24}, if its `Server` has to access the other `Server` because it does not contain the required key itself. A message is printed when the `Client` finishes {21}.

7.2.2 The Server Process

The coding for the `Server` process is shown in Listing 7-4. The `Server` process has three pairs of channels {12–17}, as can be observed from Figure 7-2. The channels `clientRequest` and `clientSend` provide the connections to the `Client` process. The channels `thisServerRequest` and `thisServerReceive` are used by this `Server` to make a request to the other `Server` and then receive a response back. The channels `otherServerRequest` and `otherServerReceive` are used by the other server to make a request to this server. Recall that we are only considering a situation in which there are only two servers. The property `dataMap` {18} holds a map of the keys and values stored in the `Server`.

```
10 class Server implements CSProcess{
11
12   def ChannelInput clientRequest
13   def ChannelOutput clientSend
14   def ChannelOutput thisServerRequest
15   def ChannelInput thisServerReceive
16   def ChannelInput otherServerRequest
17   def ChannelOutput otherServerSend
18   def dataMap = [ : ]
19
20   void run () {
21     def CLIENT = 0
```

```
22      def OTHER_REQUEST = 1
23      def THIS_RECEIVE = 2
24      def serverAlt = new ALT ([clientRequest,
25                                otherServerRequest,
26                                thisServerReceive])
27      while (true) {
28        def index = serverAlt.select()
29
30        switch (index) {
31          case CLIENT :
32            def key = clientRequest.read()
33            if ( dataMap.containsKey(key) )
34              clientSend.write(dataMap[key])
35            else
36              thisServerRequest.write(key)
37            //end if
38            break
39          case OTHER_REQUEST :
40            def key = otherServerRequest.read()
41            if ( dataMap.containsKey(key) )
42              otherServerSend.write(dataMap[key])
43            else
44              otherServerSend.write(-1)
45            //end if
```

```
46            break
47          case THIS_RECEIVE :
48            clientSend.write(thisServerReceive.read() )
49            break
50        } // end switch
51      } //end while
52    } //end run
53 }
```

Listing 7-4 The Server Process Coding

The `Server` can receive inputs from three different sources; its `Client`, a request from the other `Server` and a result from the other `Server` in response to a request made by this `Server`. This is reflected in the creation of three case constants {21-23} and an alternative over the three input channels {24–27}. The `Server` then loops over which ever alternative guard is enabled and `Select`ed followed by executing the related code body in a `switch` statement {28, 30}.

In the case of a `CLIENT` request, the requested key value is read {32} and then a test is made to see if that key is present in the `Server` {33}. If the `key` is present then value in the `dataMap` entry corresponding to the `key` value is sent back to the `Client` {34}; otherwise the request is passed to the other server {36}.

In the case of an `OTHER_REQUEST` from the other server {39–46}, the `CLIENT` code body described above is repeated except that a -1 value is returned {44} if a map entry with the requested key value is not found (This should not happen!).

Finally, in the case `THIS_RECEIVE`, which is a response to a request made by this server on the other server {47-49}, the received value is returned to the `Client` {48}.

7.2.3 Running the Network of Clients and Servers

The script used to test the client and server model is shown in Listing 7-5. The eight channels are defined {10-17}, where the notation X2Y implies that the writing (`.out()`) end of the channel is in the process represented by X and the reading (.in()) end of the channel is in the Y process.

The `map`s associated with each server are then defined {19–20}. The list of key values that each client is to access is then specified {22, 23} and it should be noted that both clients read values from both servers.

```
10 def S02S1request = Channel.one2one()
11 def S12S0send = Channel.one2one()
12 def S12S0request = Channel.one2one()
13 def S02S1send = Channel.one2one()
14 def C02S0request = Channel.one2one()
15 def S02C0send = Channel.one2one()
16 def C12S1request = Channel.one2one()
```

```
17 def S12C1send = Channel.one2one()
18
19 def server0Map = [1:10,2:20,3:30,4:40,5:50,6:60,7:70,8:80,9:90,10:100]
20 def server1Map = [11:110,12:120,13:130,14:140,15:150,
21                 16:160,17:170,18:180,19:190,20:200]
22
23 def client0List = [1,12,3,14,15,16,7,18,9,10]
24 def client1List = [11,12,13,14,15,6,17,8,19,20]
25
26 def client0 = new Client ( requestChannel: C02S0request.out(),
27                            receiveChannel: S02C0send.in(),
28                            selectList: client0List,
29                            clientNumber: 0)
30
31 def client1 = new Client ( requestChannel: C12S1request.out(),
32                            receiveChannel: S12C1send.in(),
33                            selectList: client1List,
34                            clientNumber: 1)
35
36 def server0 = new Server ( clientRequest: C02S0request.in(),
37                            clientSend: S02C0send.out(),
38                            thisServerRequest: S02S1request.out(),
39                            thisServerReceive: S12S0send.in(),
40                            otherServerRequest: S12S0request.in(),
41                            otherServerSend: S02S1send.out(),
42                            dataMap: server0Map)
43
44 def server1 = new Server ( clientRequest: C12S1request.in(),
45                            clientSend: S12C1send.out(),
46                            thisServerRequest: S12S0request.out(),
47                            thisServerReceive: S02S1send.in(),
48                            otherServerRequest: S02S1request.in(),
49                            otherServerSend: S12S0send.out(),
50                            dataMap: server1Map)
51
52 def network = [client0, client1, server0, server1]
53 new PAR (network).run()
```

Listing 7-5 The Defintion of Channels, Server Maps and Client Key Lists

The processes are then defined such that `client0` is connected to `server0` and `client1` is connected to `server1`. The process network is then defined {52} and the network run {53}.

The result, see Output 7-2, from the execution of the network, shown in Figure 7-2 and Listing 7-5, can be seen, by inspection, to have not completed because the final output line that indicates the client processes have finished and terminated is not printed.

Client 0 has 10 values in [1, 12, 3, 14, 15, 16, 7, 18, 9, 10]
Client 1 has 10 values in [11, 12, 13, 14, 15, 6, 17, 8, 19, 20]

Output 7-2 Correct Output from the Clients and Servers network

If we replace lines {23, 24} of Listing 7-5 with the following:

```
def client0List = [1,2,3,4,5,6,7,18,9,10]
def client1List = [11,12,13,4,15,16,17,18,19,20]
```

then the result shown in Output 7-3 is generated. This shows that both clients have accessed all the data they require because they have produced the final completion message. Thus the ordering of client requests is significant for the correct operation of the network of processes. This is something that should not be allowed to occur. By inspection we can see that both servers can get into a state where they are trying to access the other server either by making a request or by both waiting to receive a response so that neither of them can complete a communication.

```
Client 1 has 10 values in [11, 12, 13, 4, 15, 16, 17, 18, 19, 20]
Client 0 has 10 values in [1, 2, 3, 4, 5, 6, 7, 18, 9, 10]
Client 0 has finished
Client 1 has finished
```

Output 7-3 Deadlocked Client Server Results

The programmer should resist the temptation to insert print statements to determine what has happened because these can often remove the time critical nature of the interactions and the system appears to work. This occurs because the network of processes is sequentialised by outputting to the common console or other display device. This will be explored in the exercises. It can even depend upon the length of the print statement as to whether a non-working system can be made to work.

It should be noted also that different executions of the same network, unaltered in any way, will often produce different deadlock situations. It is dependent on the dynamics of the network. It can also vary with the combination of processor, operating system, java virtual machine and the number of cores and how they are utilised.

7.3 Summary

In this chapter we have demonstrated how deadlock can occur, first in a very simplistic manner and secondly, in a more complex set of interactions that are very hard to foresee. In the second case the programmer attempted to ensure that the servers were undertaking as many interactions as possible with the clients. A more careful programmer might have decided that instead of separating the request and response made to the other server they would ensure that the response from the other server was received and returned to the client before embarking upon another interaction. This would work in the case where both servers were executing on the same processor because the interactions would be interleaved and thus some distinct ordering might give the impression that the network operated correctly.

However, this would be a fool's paradise in the case of servers running on different machines because in due course the situation would arise where both servers were trying to send a request to each other and the deadlock would occur, possibly after a long period, which had given the impression that the system worked correctly and that the fault lay elsewhere.

Thus we have to find a design pattern that permits the safe design of parallel systems and that is the content of the next chapter.

7.4 Exercises

Exercise 71

By placing print statements in the coding for the Server and Client processes see if you can determine the precise nature of the deadlock in the Client Server system. You will probably find it useful to add a property to the Server process by which you can identify each Server.

8 Client-Server: Deadlock Avoidance by Design

The fundamental client-server design pattern is described by

- defining client behaviour
- defining server behaviour
- defining the required relationship between clients and servers to ensure deadlock and livelock freedom
- analysing the Queue processing system from Chapter 6 and showing it to comply with the design pattern
- re-implementing the multi-server system of Chapter 7 so that we can design and implement a deadlock and livelock free version

Chapter 7 demonstrated with two examples, one obvious and the other less so, that deadlocked systems can be constructed quite easily even if the thought given to the design would suggest otherwise. A design pattern is required that ensures deadlock and also livelock freedom. Brinch Hansen (Brinch Hansen, 1973) formulated a design approach for operating systems in the 1970s based upon a client-server architecture. It is a slightly updated version of that design approach that is presented here as the client-server design pattern (Welch, et al., 1993) (Martin & Welch, 1997). It is captured in two simple rules, together with a method for analysing a network.

1. A client process that issues a request to a server process guarantees to accept any response from that server immediately. A client – server interaction requires a client request upon the server but it is not necessary for there to be a communication from the server to the client process.
2. A server process that accepts a request from a client process guarantees to return a response to the client process within finite time. In addition, a server process will never send a message to any of its clients without having first received a request from a client. A server process can behave as a client to another server process.
3. Deadlock and livelock will not occur in such a network of client and server processes provided a labelling of the client and server ends of the interactions between processes does not result in a completed circuit of clients and servers.

8.1 Analysing the Queue Accessing System

The `Queue` system discussed previously in Section 6.2 is, in fact, an example of a system that implements the above set of client – server rules.

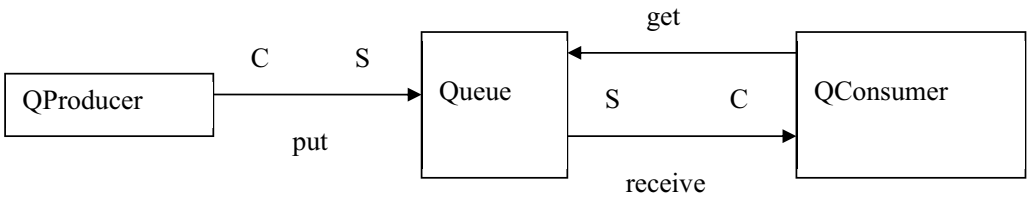


Figure 8-1 Client – Server Labelling of the Queue Processing System

The process `QProducer` acts as a client to `Queue` process, which acts as a server. This interaction does not involve a return communication from `Queue` to `QProducer`. `QConsumer` also acts as a client to the `Queue` process but in this interaction the `QConsumer` process does expect a response. The client behaviour of the `QProducer` process is captured in the following code snippet taken from Listing 5-6. The client request is captured in the `write` method on the channel `put` {2}.

```
1 for ( i in 1 .. iterations ) {
2 put.write(i)
3 }
```

Similarly, the client behaviour of `QConsumer` is shown in the following snippet and is taken from Listing 5-7. The client request is captured in the `write` on the `get` channel {5}, which can be simply interpreted as a signal for the `Queue` process. As a client the `QConsumer` must be ready to receive a response from the `Queue` process as soon as it is available. This is simply achieved by `read`ing the response on the `receive` channel {6}. Crucially, no other processing takes place between the request {5} and the response {6}.

```
4 while (running) {
5 get.write(1)
6 def v = receive.read()
7 }
```

The `Queue` process, see Listing 5-8, simply alternates over the `put` and `get` channels and thus can never generate an output of its own accord and thus behaves as a server for both its interactions. The network, shown in Figure 8-1, contains no circuits and thus is guaranteed to be free from deadlock and livelock.

8.2 Client and Server Design Patterns

The behaviour of processes that implement the Client and Servers patterns is given in the following design templates.

```
1 class ClientTemplate implements CSProcess {
2   def ChannelOutput request
3   def ChannelInput response            // may not be required
4   void run() {
5     // initialise
6     while (true) {
7       // create server request object
8       request.write ( requestObject ) // could be a signal
9       result = response.read()     // may not be required
10      // process result
11    }
12  }
13 }
```

Template 8-1 Client Design Template

A process that behaves as a client (Template 8-1) will have an output channel upon which it makes `request`s to its server {2}. It will probably have an input channel upon which it receives a `response` {3} from the server but this may not always be necessary. A client process may undertake some initialisation {5} before entering the main loop of the process {6}. Depending upon the nature of the interaction the client process will either create a `requestObject` {7} or cause a signal to be sent to the server {8} if no explicit data is required for the server to respond to the client process. The client process will immediately wait for the response from the server if there is one {9}. The process will then continue processing.

The server template (see Template 8-2) indicates that a server process requires a `request` {2} input channel and may have a `response` {3} output channel if there is an explicit result written to the requesting client process. The server may undertake some initialisation {5} after which it enters the main loop of the process {6}. The server responds to client requests by either reading some form of `requestObject` or a signal {7}. The nature of the request is determined unless that is implicit as a result of receiving a signal {8}. The server then determines the result, which may require access to another server {9} after which the result, if any, is written to the client {10}. The server then may have to update some internal state {11} before repeating the loop.

```
1  class ServerTemplate implements CSProcess {
2    def ChannelInput request
3    def ChannelOutput response // may not be required
4    void run() {
5      // initialise
6      while (true) {
7        def requestObject = request.read() // may be a signal
8        // process requestObject
9        // determine any result, may require request to another server
10       response.write(result) // may not be required
11       // update any internal state
12     }
13   }
14   }
```

Template 8-2 Server Design Template

By inspection we can determine that the client template does implement the first behavioural requirement for a client, given previously, in that once it has made a request on a server it is immediately ready to receive any response from that server. Similarly the server process template shows that the server will respond in finite time. Two cases need to be considered. If the server makes no request to another server then the response must be fully determined within the server process and thus this can be completed in finite time as there will be no other communication, which is the only source of indefinite delay, provided the computation is finite. If the server process makes a client style request on another server then provided the requested server maintains the client – server contract then the originating server can respond in finite time.

8.3 Analysing the Crossed Servers Network

Figure 8-2 shows the client-server labelling of the network shown previously in Figure 7-2. The relationship between the Client and Server processes is not the problem and in fact by inspection of the Client process, Listing 7-3 lines {23–24}, it can be seen that this process does in fact implement client behaviour.

The problem lies with the `Server` processes, where there is a circuit from one `Server` to the other and back again. Even if we implemented the `Server`, Listing 7-4, so that it did not interleave access, it would still deadlock. In the implementation, shown in Listing 7-4, the chance of deadlock occurring is compounded because the `Server` does not wait for a response from the other `Server` but starts another client interaction. A simplistic solution could be attempted by making the Server process act as a client when it is accessing the other Server. This will decrease the incidence of deadlock but it will still happen, but less frequently, which perhaps makes it even more annoying for the `Client` processes. Thus a different solution has to be found.

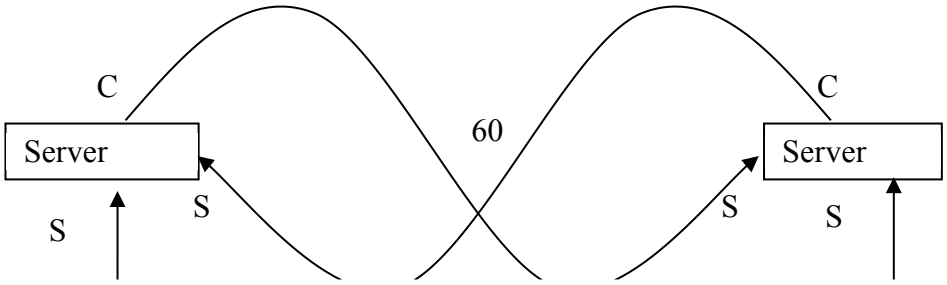


Figure 8-2 Client – Server Labelling of the Crossed Server Network

8.4 Deadlock Free Multi-Client and Servers Interactions

Figure 8-3 shows a solution to the problem that is achieved by the use of a multiplexer. A multiplexer is a process that accepts inputs from a number of input channels and then outputs these input communications on a single output channel. A set of simple multiplexers is available in the package `org.jcsp.groovy.utils`.

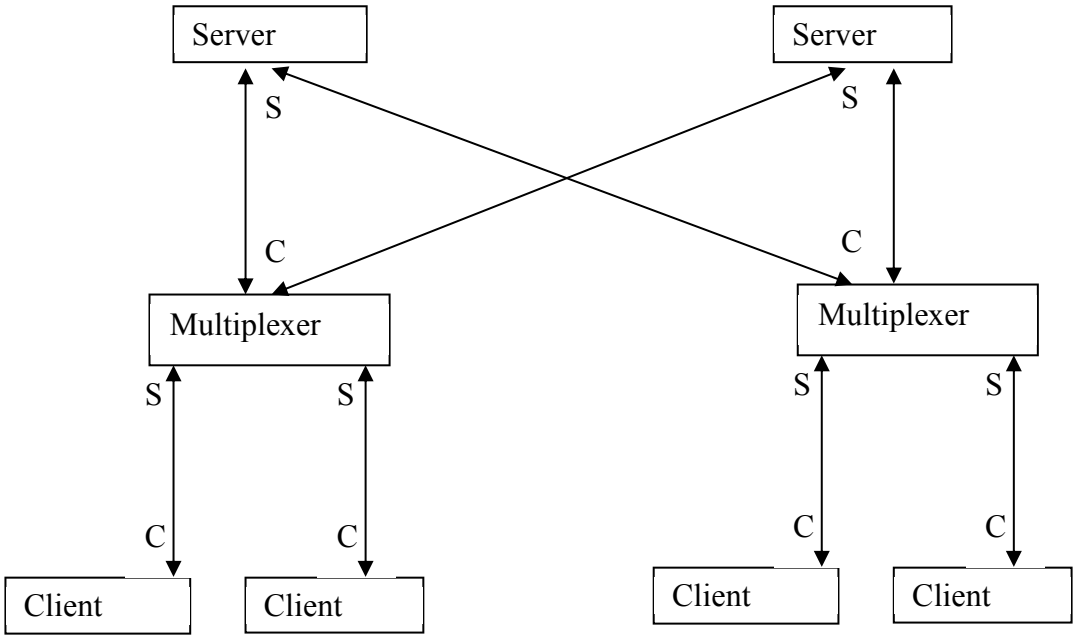


Figure 8-3 Multi-Client and Server Network

A Client makes a request upon a Multiplexer behaving as a server, which contains the data required to determine upon which Server the required data resides. The Multiplexer then behaves as a client and makes a request to the required Server. The corresponding data is then returned from the Server to the Multiplexer and then finally the Client receives the data it requested. By inspection of the network we can see there are no circuits of client – server labelling and hence the network will be deadlock free, provided the processes are implemented in a manner that respects the rules of the client-server design pattern.

8.4.1 The Multiplexer Process

Listing 8-1 shows the multiplexer used in this system. `CSMux` is more complex than the simple multiplexer concept described previously. Requests are received from `Client`s by `CSMux` behaving as a server and then `CSMux` determines the `Server` upon which the required data is to be found. The request is then forwarded to a `Server` by the `CSMux` behaving as a client. `CSMux` waits for the response from a `Server` that it then returns to the original `Client` behaving as a server. `CSMux` utilises properties of type `ChannelInputList` and `ChannelOutputList`. These are two helper classes created as part of the Groovy Parallel capability. As their names suggest these provide lists of channel input ends and channel output ends respectively.

```
10 class CSMux implements CSProcess {
11
12   def ChannelInputList inClientChannels
13   def ChannelOutputList outClientChannels
14   def ChannelInputList fromServers
15   def ChannelOutputList toServers
16   def serverAllocation = [ ]
17
18   void run() {
19     def servers = toServers.size()
20     def muxAlt = new ALT (inClientChannels)
21     while (true) {
22       def index = muxAlt.select()
23       def key = inClientChannels[index].read()
24       def server = -1
25       for ( i in 0 ..< servers) {
26         if (serverAllocation[i].contains(key)) {
27           server = i
28           break
29         }
30       }
31       toServers[server].write(key)
32       def value = fromServers[server].read()
33       outClientChannels[index].write(value)
34     }
35   }
36 }
```

Listing 8-1 The Multiplexer Coding

The `ChannelInputList inClientChannels` {12} is a list of input channel ends from each of the `Client`s connected to `CSMux`. Similarly, `fromServers` {14} is a list of the channel input ends coming from each of the `Server`s connected to `CSMux`. The list of out channels ends that connects `CSMux` to its `Client`s is contained in the property `outClientChannels` {13} and the outputs from `CSMux` to the connected `Server`s is passed as property `toServers` {15}. The property `serverAllocation` is a `List` of `List`s such that each internal `List` contains the keys of the values held respectively in each `Server`. There is one list element per `Server` in `serverAllocation` {16}. The `size()` method is used to find the number of `Servers` {19} because there must be as many `Server` processes as there are channels in `toServers`. The alternative, `muxAlt` {20}, is simply constructed from the `inClientChannels ChannelInputList`.

Within the loop {21–34}, the `index` of the enabled alternative is `select`ed {22} and its value used to read a `key` value from the corresponding element of `inClientChannels` {23}. The `for` loop {25–30} is used to determine in which `server` the `key` is located. The value of the `key` is written to the `server` element of the `ChannelOutputList toServers` {31}. As this is the start of a client style interaction the `value` corresponding to the `key` is `read`, as soon as it is available on the `server` element of the `ChannelInputList fromServers` {32}. This maintains the client-server relationship between `CSMux` and the `Server`. The `value` is then written to the `index` element of `outClientChannels` {33} thereby completing the server style interaction between `CSMux` and the originating `Client` process.

This interaction typifies a more complex client and server interaction whereby the client makes a request on a server style process which then becomes a client to another server. This can be undertaken as many times as the application requires and, provided there are no circuits in the clients and servers, is guaranteed to be deadlock and livelock free, provided the processes implement the client and server behaviours as defined previously.

8.4.2 The Server Process

The coding of the `Server` process is shown in Listing 8-2. A `Server` has two channel list properties, one, `fromMux` is the input channels from `CSMux` {12} and the other {13}, `toMux`, provides the output channels to `CSMux`. The property `dataMap` {14} is used to hold the `Map` of keys and values held within this `Server`. An alternative `serverAlt` is used to alternate over the `fromMux` input channels {17}. Once an enabled alternative has been selected {20}, its `index` is used to read the `key` value from the corresponding element of `fromMux` {21}. This `key` value is then used to access `dataMap`, the value of which is written to the related `CSMux` process using the `index` element of `toMux` {22}.

This simple interaction implements the simplest form of server behaviour, whereby the server reads the request and responds immediately to the request with the required data value.

```
10 class Server implements CSProcess{
11
12   def ChannelInputList fromMux
13   def ChannelOutputList toMux
14   def dataMap = [ : ]
15
16   void run() {
17     def serverAlt = new ALT(fromMux)
18
19     while (true) {
20       def index = serverAlt.select()
21       def key = fromMux[index].read()
22       toMux[index].write(dataMap[key])
23     }
24   }
25 }
```

Listing 8-2 The Server Process Definition

8.4.3 Exercising the System of Clients and Servers

Listing 8-3 gives the script that causes the system of `Clients`, `CSMux` processes and `Servers` to be invoked and permit the number of `Clients` per `CSMux` to be varied at run time. The number of `Servers` is limited to 2, identified as `Server` zero and `Server` one. Similarly there are two `CSMux` processes providing the multiplex capability, referred to as `CSMux` zero and one respectively. In particular, it can be seen that the `Client` process definition used in Chapter 7 is reused {10}, further reinforcing that it already implemented the required client behaviour.

Initially the number of `clients` {11} is obtained and then used together with `servers` {12} to create a set of `One2OneChannel` arrays {14–21}. The naming convention uses C to refer to a `Client` connection, M a `CSMux` connection and S a `Server` connection. Thus, `M0ToC0` {15} provides the connection from `CSMux` zero to the `Clients` attached to that multiplexer and `M1ToS` connects `CSMux` one to both `Servers`.

These channel arrays are then converted to instances of `ChannelInputList` and `ChannelOutputList` by simply calling the constructor of the required class {23–30}. For the channels lists that provide the cross connections between the `Server` and `CSMux` processes we first create new, empty, instances of the necessary channel lists, to which the required channel elements are `append`ed {32–46}. The regularity of the coding arises because all the elements at one end of a channel list have to be allocated to different processes whereas they are all accessed by a single process at the other end.

```
10 import c07.Client
11 def clients = Ask.Int ("Number of clients per server; 1 to 9 ? ", 1, 9)
12 def servers = 2
13
14 def C0ToM0 = Channel.one2oneArray (clients)
15 def M0ToC0 = Channel.one2oneArray (clients)
16 def C1ToM1 = Channel.one2oneArray (clients)
17 def M1ToC1 = Channel.one2oneArray (clients)
18 def M1ToS = Channel.one2oneArray (servers)
19 def M0ToS = Channel.one2oneArray (servers)
20 def S0ToM = Channel.one2oneArray (servers)
21 def S1ToM = Channel.one2oneArray (servers)
22
23 def clientsToM0 = new ChannelInputList (C0ToM0)
24 def clientsToM1 = new ChannelInputList (C1ToM1)
25 def M0ToClients = new ChannelOutputList(M0ToC0)
26 def M1ToClients = new ChannelOutputList(M1ToC1)
27 def Mux0ToServers = new ChannelOutputList(M0ToS)
28 def Mux1ToServers = new ChannelOutputList(M1ToS)
29 def Server0ToMuxes = new ChannelOutputList (S0ToM)
30 def Server1ToMuxes = new ChannelOutputList (S1ToM)
31
32 def Server0FromMuxes = new ChannelInputList()
33 Server0FromMuxes.append(M0ToS[0].in())
34 Server0FromMuxes.append(M1ToS[0].in())
35
36 def Server1FromMuxes = new ChannelInputList()
37 Server1FromMuxes.append(M0ToS[1].in())
38 Server1FromMuxes.append(M1ToS[1].in())
39
40 def Mux0FromServers = new ChannelInputList ()
41 Mux0FromServers.append(S0ToM[0].in())
42 Mux0FromServers.append(S1ToM[0].in())
43
44 def Mux1FromServers = new ChannelInputList ()
45 Mux1FromServers.append(S0ToM[1].in())
46 Mux1FromServers.append(S1ToM[1].in())
47
48 def server0Map = [1:10, 2:20, 3:30, 4:40, 5:50,
49                   6:60, 7:70, 8:80, 9:90, 10:100]
50 def server1Map = [11:110,12:120,13:130,14:140,15:150,
51                   16:160,17:170,18:180,19:190,20:200]
52 def serverKeyLists = [ [1,2,3,4,5,6,7,8,9,10],
53                        [11,12,13,14,15,16,17,18,19,20] ]
54
55 def client0List = [1,12,3,14,15,16,7,18,9,10]
56 def client1List = [11,12,13,14,15,6,17,8,19,20]
```

```
57
58 def network = [ ]
59 def server0ClientList = (0 ..< clients).collect { i ->
60                             return new Client ( requestChannel:
                               C0ToM0[i].out(),
61                                    receiveChannel: M0ToC0[i].in(),
62                                    clientNumber: i,
63                                    selectList: client0List)
64                                    }
65 def server1ClientList = (0 ..< clients).collect { i ->
66                   return new Client ( requestChannel: C1ToM1[i].out(),
67                                       receiveChannel: M1ToC1[i].in(),
68                                       clientNumber: i+10,
69                                       selectList: client1List)
70                                       }
71 network << new CSMux ( inClientChannels: clientsToM0,
72                        outClientChannels: M0ToClients,
73                        fromServers: Mux0FromServers,
74                        toServers: Mux0ToServers,
75                        serverAllocation: serverKeyLists)
76 network << new CSMux ( inClientChannels: clientsToM1,
77                        outClientChannels: M1ToClients,
78                        fromServers: Mux1FromServers,
79                        toServers: Mux1ToServers,
80                        serverAllocation: serverKeyLists)
81 network << new Server ( fromMux: Server0FromMuxes,
82                        toMux: Server0ToMuxes,
83                        dataMap: server0Map)
84 network << new Server ( fromMux: Server1FromMuxes,
85                        toMux: Server1ToMuxes,
86                        dataMap: server1Map)
87 new PAR(network + server0ClientList + server1ClientList).run()
```

Listing 8-3 Script to Run the Network of Clients and Servers

This is followed by the definition of the `Map` data structures {48–51} that are passed to each `Server` instance as the property `dataMap` {Listing 8-2, 14}. The `serverKeyLists` {52–53} comprise the sets of key values associated with each Server and are passed to each `CSMux` as property `serverAllocation` {Listing 8-1, 16}. Similarly, the list of key values which each `Client` process is to access is defined separately for the `Clients` connected to each `CSMux` process {55, 56}. These `Lists` are passed to a Client process as property `selectList` {Listing 7-3, 15}.

The network of processes required to run the system is then created. The Groovy `collect` method is used to construct a list of processes that are returned as new `Client` process instances in the associated closure. Two such `List`s are required one for each set of `Client`s attached to each of the `CSMux` processes {59–70}. In the definition of a new `Client` note how the individual elements of the channel arrays are accessed and further note that the ends of the channels so referenced are not part of the previously defined channel lists.

The empty `List network` {58} is then populated with the required instances of the `CSMux` and `Server` processes {71-86} using the append (<<) operator. Finally, a `PAR` is invoked {87} by passing the sum of all the process lists as its parameter, which can then be `run()`. The output from an execution of the network is analysed by ensuring that the Server `dataMap`s are accessed in the order specified in `client0List` and `client1List`. It should be noted that the Client processes attached to CSMux zero are numbered from 0 and those to CSMux one from 10. It can be observed that all the Clients access all the required elements of the servers in the order specified. If the system is executed with one `Client` per `CSMux;` then the version that deadlocked in Chapter 7 can be seen to be operating entirely as expected because the elements accessed from each server are those that deadlocked previously. The major change is that we can now run multiple clients per server, thereby increasing the complexity of the interactions that are being undertaken.

8.5 Summary

In this chapter the concept of the client-server design pattern has been introduced. This is the most important design pattern we shall use and will become fundamental to all the subsequent designs used in the rest of the book.

The key aspect to assimilate is that design is initiated by a network diagram showing the processes, their connections and the data that flows between them along the channels. The diagram is then analysed from the point of view of any client-server interactions and adjustments made to ensure deadlock and livelock freedom. Finally, the code for each of the processes is produced ensuring that it maintains the required client and server behaviour defined in the diagram.

8.6 Exercises

Exercise 8-1

Modify the `Client` process `c07.Client` so that it can ensure that the values returned from the `Server` arrive in the order expected according to their `selectList` property. It should print a suitable message that the test has been undertaken and whether it passed or failed. You are **not** to use the `GroovyTestCase` mechanism because this would require that the `CSMux` and `Server` processes would have to terminate, which would require a lot of unnecessary programming. Hint: you can use the relationship between key and values held in the database as a means of testing the returned values.

9 External Events: Handling Data Sources

External event handling is shown to be very amenable to concurrent and parallel programming by:

- managing events as communications rather than interrupts
- exploitiong the client-server pattern to build architectures that embody event handling
- separating event handling from processing using a buffer that is a pure server, thereby ensuring deadlock freedom
- enabling analysis of system behaviour
- permitting determination on the time upper bound for event procesisng
- processing events from many different sources

Traditionally, real-time systems that respond to external stimuli have utilised interrupts. An interrupt is a hardware signal that indicates that a device needs to be serviced and which causes the processors' central processing unit to interrupt the current program and invoke the device's service routine returning to the original program once the device has been serviced. Over the years a great deal of effort has been expended in trying to make interrupt based systems more efficient and easier to program. However, the basic problem still remains that an interrupt causes the halting of the current program, saving its state and then starting an interrupt service routine. The problems become more complex when an interrupt service routine is itself interrupted by a device with a higher priority. The approach has been to reduce the amount of time when interrupts are disabled. This in itself leads to further problems because it is very difficult then to foresee the precise nature of interactions between interrupts that can then take place. It is these indeterminate interactions that cause problems when systems are running because it is impossible to test for all the possible interactions, especially in highly complex systems.

The framework built so far, using parallelism and alternation to capture non-deterministic behaviour, provides a means of describing, implementing and analysing such event driven systems. Rather than building a system that interrupts itself on receipt of an event notification; build a system that expects such events to occur so that programmers can better reason about its behaviour. In effect, the external event is considered to be the same as a channel communication. Furthermore, the client-server design pattern gives us a handle by which the system can be analysed to ensure that unwanted interactions between events do not occur.

9.1 An Event Handling Design Pattern

The aim of the pattern is to allow the system to respond to external events as quickly as possible. However, the situation has to be considered that events may occur so rapidly that the system cannot deal with all the events. Such a situation tends to overwhelm interrupt based systems. The pattern also has to take account of any priority requirement the application may have, thereby influencing the order in which events are handled. Such ordering of the handling of events may result in some events being lost. However, if the designer is aware of this situation then steps can be taken at design time to ameliorate their effects.

The key to building an event handling system is that the process dealing with receipt of the event has to be ready, waiting, for the associated channel (event) communication, so it can be read and the associated data passed on to another process. The event receiving process can then return to the state of waiting for the next event communication. If we connect the event receiving process directly to the event processing process then the event receiver might be delayed by having to wait for the processing of another event to finish. We thus require an intermediate stage that separates event receiving from event processing. This can be implemented by some form of buffer. More specifically, the buffer should always be ready to receive a communication from the event receiver process. This may mean that previous buffered values may be overwritten. In addition, a mechanism by which buffered values can be requested from the buffer process, in a manner similar to that used in the Queue process described in Chapter 5. The resulting process structure is shown in Figure 9-1, to which a client-server labelling has been added. It demonstrates that there are no client-server loops in the architecture.

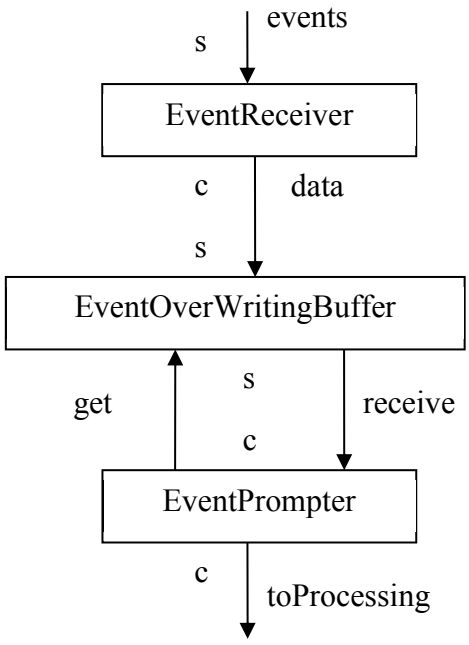


Figure 9-1 Event Handling Design Pattern

Events are received by the EventReceiver and immediately sent to the EventOverwritingBuffer so that EventReceiver is ready to read the next event. The EventPrompter indicates that it wants to get some data, which it will receive immediately from the EventOverwritingBuffer if data is already buffered or it will have to wait until an event has been input. The EventPrompter then writes the data to the rest of the system where it is processed. Thus it is EventPrompter that has to wait until subsequent processing can be undertaken, allowing EventReceiver always ready to read an event. Later we shall show that the time required to process events can in fact be calculated to give an absolute upper bound on the performance of the system. Such a bound cannot be calculated for interrupt based systems. In addition, the client-server labelling shows that the pattern has no deadlock or livelock inherent within it and thus provided the rest of the system is also deadlock and livelock free ensures that the system will behave as expected. This is easily deduced because the EventOverwritingBuffer is a pure server and hence any client-server circuit cannot exist.

9.2 Utilising the Event Handing Pattern

The pattern can be transformed easily into a set of processes that achieves its effect.

9.2.1 The EventReceiver Process

Listing 9-1 shows the definition of the `EventReceiver` process. The process has `eventIn` {12} and `eventOut` {13} channel properties. In this implementation, every input that is `read` from `eventIn` is immediately `writ`ten to the `eventOut` channel {17}. In a realistic implementation it would probably be necessary to determine the source of the event and possibly read some data from a hardware register. However, we can presume that such additional processing would not create any substantial delay within the system because the event would not be raised if there was no reason.

```
10 class EventReceiver implements CSProcess {
11
12   def ChannelInput eventIn
13   def ChannelOutput eventOut
14
15   void run() {
16     while (true){
17       eventOut.write(eventIn.read())
18     }
19   }
20 }
```

Listing 9-1 The EventReceiver Process

9.2.2 The Event Overwriting Buffer Process

The implementation of EventOWBuffer is shown in Listing 9-2.

```
10 class EventOWBuffer implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelInput getChannel
14   def ChannelOutput outChannel
15
16   void run () {
17     def owbAlt = new ALT ( [inChannel, getChannel] )
18
19     def INCHANNEL = 0
20     def GETCHANNEL = 1
21     def preCon = new boolean[2]
22     preCon[INCHANNEL] = true
23     preCon[GETCHANNEL] = false
24     def e = new EventData ()
25     def missed = -1
26
27     while (true) {
28       def index = owbAlt.priSelect ( preCon )
29
```

```
30        switch ( index ) {
31          case INCHANNEL:
32            e = inChannel.read().copy()
33            missed = missed + 1
34            e.missed = missed
35            preCon[GETCHANNEL] = true
36            break
37
38          case GETCHANNEL:
39            def s = getChannel.read()
40            outChannel.write ( e )
41            missed = -1
42            preCon[GETCHANNEL] = false
43            break
44
45        } // end switch
46      } // end while
47    } // end run
48 }
```

Listing 9-2 The EventOWBuffer Process

The channel `inChannel` {12} inputs data from the `EventReceiver` process. The `getChannel` {13} receives a signal from `EventPrompter` whenever that process requires data. The response to `EventPrompter` is to output event data on the channel `outChannel` {14}. The process receives inputs on its input channels over which it must alternate as the order in which such inputs are read cannot be determined. This is captured in the definition of `owbAlt` {17}. The `EventOWBuffer` also has to capture the behaviour that requests for data from the `EventPrompter` process can only be allowed when the buffer contains data. To this end we used pre-conditions on `owbAlt` in a manner similar to that used in the `Queue` process described in Chapter 5. The constants `INCHANNEL` {19} and `GETCHANNEL` {20} are used to access the elements of the `preCon` {21} `boolean` array and also to identify the `case`s within the `switch` that implements the main processing loop. The initial values of the `preCon` elements can be specified as follows. The process is always willing to accept inputs on its `inChannel` and thus this element is always `true` {22}. Initially there is no data in the buffer and thus requests to get data from `EventPrompter` must not be permitted and thus that pre-condition has to be set `false` {23}. The actual buffer is represented by the variable `e` {24} and is of type `EventData`, see Listing 9-4. The variable `missed` {25} will count the number of times the data in the buffer `e` was overwritten and will be passed through the system so that its performance can be analysed. It is initialised to -1 so that when the next event is read its value will be considered not to have been overwritten because the value of `missed` will then be 0.

The main loop of EventOWBuffer {27–46} initially determines the `index` of the enabled channel, with priority being given to `inChannel` {17} because we always want `EventReceiver` to be ready to read the next event. In that case, the event data is read from `inChannel` {32} and a deep copy is made into the buffer variable `e`. The interface `JCSPCopy`, defined in `org.jcsp.groovy`, defines an abstract method `copy`() that can be used to make a deep copy of an object. Recall that if an object is transferred from one process to another then if these processes are on the same processor then this communication is achieved by passing an object reference. We must ensure that two processes do not access the same object at the same time and hence the need to make a deep copy of the object. The value of `missed` is incremented {33} and saved in the buffer variable `e` {34}. The `preCon` element `GETCHANNEL` can now be set `true` {35} because there is data in the buffer that can be sent to `EventPrompter` following a request for data.

Once the buffer contains data then requests for data can be read from the `getChannel` {39} and the contents of the buffer are immediately written to the `outChannel` {40}. This interaction ensures the process behaves like a server. The `preCon` element `GETCHANNEL` must now be set `false` {42} because there is no longer any data in the buffer and likewise the variable `missed` must be reset to -1 {41}.

9.2.3 The Event Prompter Process

This process is shown in Listing 9-3. This process has channel properties {12–14} that reflect the process structure shown in Figure 9-1.

```
10 class EventPrompter implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput getChannel
14   def ChannelOutput outChannel
15
16   void run () {
17     def s = 1
18     while (true) {
19       getChannel.write(s)
20       def e = inChannel.read().copy()
21       outChannel.write( e )
22     }
23   }
24 }
```

Listing 9-3 The EventPrompter Process

A signal is written to the `getChannel` {19}, the completion of which may be delayed until the `EventOWBuffer` contains event data. The response from `EventOWBuffer` is immediately read into a variable `e` {20} and also uses the `copy()` method to ensure that the data cannot be modified as it resides within the `EventPrompter` before being output to the next process. The data is then written {21} to the `outChannel`, where yet again a delay may be incurred due to the processing system not being in a state where the data from this event source can be processed.

9.2.4 The EventData Class

`EventData` contains three properties for this explanatory description comprising `source` {12}, `data` {13} and `missed` {14}, shown in Listing 9-4. The `source` is used to indicate from which event source the event came. The actual data value sent by the event is contained in `data`. The class implements the `Serializable` interface so that `EventData` objects can be communicated over networks. The interface `JCSPCopy` is implemented so that the `copy` method {16–21} can be defined that makes a deep copy of `EventData` objects. A `toString` method has also been provided so that event data can be more easily output {23–29}.

```
10 class EventData implements Serializable, JCSPCopy {
11
12   def int source = 0
13   def int data = 0
14   def int missed = -1
15
16   def copy() {
17     def e = new EventData ( source: this.source,
18                             data: this.data,
19                             missed: this.missed )
```

```
20      return e
21    }
22
23    def String toString() {
24      def s = "EventData -> [source: "
25      s = s + source + ", data: "
26      s = s + data + ", missed: "
27      s = s + missed + "]"
28      return s
29    }
30
31 }
```

Listing 9-4 The EventData Class Definition

9.2.5 The EventHandler Process

The `EventHandler` process is the parallel composition of the the processes shown in Figure 9-1, as shown in Listing 9-5.

The Event Handler process is written to accept input events on its `inChannel` {12}. The events are then output on its `outChannel` {13}. The process has three internal channels, `get`, `transfer` and `toBuffer` {16–18}. These are used to connect the processes that are created in the `HandlerList` {20–28}. The processes used in the `HandlerList` have been previously described in Sections 9.2.1 to 9.2.3. The processes are then invoked by a `PAR` {29}.

```
10 class EventHandler implements CSProcess {
11
12    def ChannelInput inChannel
13    def ChannelOutput outChannel
14
15    void run () {
16      def get = Channel.one2one()
17      def transfer = Channel.one2one()
18      def toBuffer = Channel.one2one()
19
20      def handlerList = [ new EventReceiver ( eventIn: inChannel,
21                                              eventOut: toBuffer.out()),
22                          new EventOWBuffer ( inChannel: toBuffer.in(),
23                                              getChannel: get.in(),
24                                              outChannel: transfer.out() ),
25                          new EventPrompter ( inChannel: transfer.in(),
26                                              getChannel: get.out(),
27                                              outChannel: outChannel )
28                          ]
29      new PAR ( handlerList ).run()
30    }
31 }
```

Listing 9-5 The EventHandler Process Definition

9.3 Analysing Performance Bounds

The ability of the design pattern to handle repeated events can be determined for two different cases. The first and simplest case occurs when there is no outstanding request for data from the `EventPrompter` process. The time to handle an event can be calculated by adding together the processing times of lines Listing 9-1 17, and Listing 9-2 lines 32–35 plus the time to undertake a single communication from `EventReceiver` to `EventOWBuffer`. This value can be determined by calculation if the time to execute each statement can be determined.

The second case is slightly more complex and concerns the situation when `EventOWBuffer` has just accepted a request to get data from `EventPrompter` and an event arrives at `EventReceiver`. The consequent processing delay comprises Listing 9-2 lines 39–42 and Listing 9-3 line 20 plus the time taken to undertake a single communication from `EventOWBuffer` to `EventPrompter`. Thus this time, plus the time to actually process the event, which is the same as the first case, gives the total time that is required to handle an event. This therefore gives an upper bound for the time to process an event and thus the maximum rate at which events can be handled in the worst case scenario. On a modern processor these times will be measured in nanoseconds. The fact that the processing system might not be able to keep up with such a rate merely points to a possible deficiency in the system design and not a failure of the ability to use parallel processing techniques to handle events.

9.4 Simple Demonstration of the Event Handling System

The demonstration comprises an `EventHandler` process which is fed with 'events' by an `EventGenerator` process that outputs data values according to a uniformly distributed delay strategy. The `EventHandler` outputs its 'events' to another process that simulates the time it takes to process an event according to a different uniformly distributed delay strategy. Finally, the processed 'events' are printed using a GPrint process.

9.4.1 The Event Generator Process

The `EventGenerator` process, shown in Listing 9-6, itself comprises two parallel processes, `EventStream` and `UniformlyDistributedDelay`. The properties of the process are passed directly to these processes and will thus be described in the next sections.

```
10 class EventGenerator implements CSProcess {
11
12   def ChannelOutput outChannel
13   def int source = 0
14   def int initialValue = 0
15   def int minTime = 100
16   def int maxTime = 1000
17   def int iterations = 10
18
```

```
19    void run () {
20      def es2udd = Channel.one2one()
21      println "Event Generator for source $source has started"
22
23      def eventGeneratorList = [
24            new EventStream ( source: source,
25                              initialValue: initialValue,
26                              iterations: iterations,
27                              outChannel: es2udd.out() ),
28            new UniformlyDistributedDelay ( minTime: minTime,
29                                            maxTime: maxTime,
30                                            inChannel: es2udd.in(),
31                                            outChannel: outChannel )
32            ]
33
34      new PAR (eventGeneratorList).run()
35    }
36 }
```

Listing 9-6 The EventGenerator Process

9.4.2 The Event Stream Process

Listing 9-7 shows the `EventStream` process, in which the `source` property {12} is used to identify the stream and which has an `initialValue` {13}. The process will output a stream of length `iterations` {14}. The stream of 'events' will be output on the channel `outChannel` {15}. The process uses the `upto` method to create the loop {20}. An event `e` is constructed {21} and then written to `outChannel` {22}. On completion the process outputs a message {25} as this will prove invaluable in understanding how the system functions.

```
10 class EventStream implements CSProcess {
11
12   def int source = 0
13   def int initialValue = 0
14   def int iterations = 10
15   def ChannelOutput outChannel
16
17   void run () {
18     def i = initialValue
19
20     1.upto(iterations) {
21       def e = new EventData ( source: source, data: i )
22       outChannel.write(e)
23       i = i + 1
24     }
25     println "Source $source has finished"
26   }
27 }
```

Listing 9-7 The EventStream Process

9.4.3 The Uniformly Distributed Delay Process

The `UniformlyDistributedDelay` process, shown in Listing 9-8, uses a random number generator {19} to produce a `delay` between `minTime` and `maxTime` {23}. The event data is read from `inChannel` {22} and after waiting for the `delay` {24} period it is output on `outChannel` {25}.

```
10 class UniformlyDistributedDelay implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14   def int minTime = 100
15   def int maxTime = 1000
16
17   void run () {
18     def timer = new CSTimer()
19     def rng = new Random()
20
```

```
21      while (true) {
22        def v = inChannel.read().copy()
23        def delay = minTime + rng.nextInt ( maxTime - minTime )
24        timer.sleep (delay)
25        outChannel.write( v )
26      }
27    }
28 }
```

Listing 9-8 The UniformlyDistributedDelay Process

The effect of the `UniformlyDistributedDelay` process is to ensure that events are generated with delays that vary uniformly between `minTime` and `maxTime`.

9.4.4 Demonstration of a Single Stream Event Processing System

The script that invokes the system with a single source of events is shown in Listing 9-9. The collection of processes comprises the processes already described, executed in parallel. An additional `UniformlyDistributedDelay` process has been included to represent the varying time it takes to process an event. The events are passed to a `GPrint` process where they are simply printed. Of particular interest is the number of events that are missed.

```
10 def eg2h = Channel.one2one()
11 def h2udd = Channel.one2one()
12 def udd2prn = Channel.one2one()
13 def eventTestList = [
14         new EventGenerator ( source: 1,
15                              initialValue: 100,
16                              iterations: 100,
17                              outChannel: eg2h.out(),
18                              minTime: 100,
19                              maxTime:200 ),
20
21         new EventHandler ( inChannel: eg2h.in(),
22                              outChannel: h2udd.out() ),
23
24         new UniformlyDistributedDelay ( inChannel:h2udd.in(),
25                                         outChannel: udd2prn.out(),
26                                         minTime: 1000,
27                                         maxTime: 2000 ),
28
29         new GPrint ( inChannel: udd2prn.in(),
30                      heading : "Event Output",
31                      delay: 0)
32         ]
33
34 new PAR ( eventTestList ).run()
```

Listing 9-9 The Srcript Used to Invoke the Single Stream Event Handling System

The events are generated with a delay that varies between 100 and 200 milliseconds {18, 19}. The simulation of processing time {26, 27} varying between 1000 and 2000 milliseconds means we would expect around 8 or 9 events to be missed but will depend on the actual random values. A sample output from the system is shown in Output 9-1.

The first two events pass through the system without any delay because that is the time when the buffers within the system are being filled. Thereafter, data appears with varying numbers of events missed and in general these match what would be expected. The last three events are produced after the event generator has finished because they are buffered up within the system. It should be noted that a check of correctness of operation is possible because the data value, after the first, is equal to the previous output data value plus the number missed plus 1.

```
Event Output
Event Generator for source 1 has started
EventData -> [source: 1, data: 100, missed: 0]
EventData -> [source: 1, data: 101, missed: 0]
EventData -> [source: 1, data: 110, missed: 8]
EventData -> [source: 1, data: 122, missed: 11]
EventData -> [source: 1, data: 128, missed: 5]
EventData -> [source: 1, data: 140, missed: 11]
```

```
EventData -> [source: 1, data: 149, missed: 8]
EventData -> [source: 1, data: 159, missed: 9]
EventData -> [source: 1, data: 168, missed: 8]
EventData -> [source: 1, data: 176, missed: 7]
Source 1 has finished
EventData -> [source: 1, data: 186, missed: 9]
EventData -> [source: 1, data: 195, missed: 8]
EventData -> [source: 1, data: 199, missed: 3]
```

Output 9-1 A Sample Output from the Event Handling System

9.5 Processing Multiple Event Streams

Now that we have seen how to process a single stream of events; it becomes noteworthy to process multiple streams of events that are generated with varying uniformly distributed delays. We combine the event generation and event handling into a single process called `EventSource` shown in Listing 9-10.

```
10 class EventSource implements CSProcess {
11
12   def source
13   def iterations = 99
14   def minTime = 100
15   def maxTime = 250
16   def ChannelOutput outChannel
17
18   void run() {
19     def eg2h = Channel.one2one()
20     def sourceList = [ new EventGenerator ( source: source,
21                                             initialValue: 100 * source,
22                                             iterations: iterations,
23                                             minTime: minTime,
24                                             maxTime: maxTime,
25                                             outChannel: eg2h.out()),
26                        new EventHandler ( inChannel: eg2h.in(),
27                                           outChannel: outChannel)
28                        ]
29
30     new PAR (sourceList).run()
31   }
32 }
```

Listing 9-10 The Process that Generates Events and Handles Them

The `EventSource` process uses the processes `EventGenerator` and `EventHandler`, previously described in Sections 9.4.1 and 9.2.5. Each source can be assigned its own `minTime` {14, 23} and `maxTime` {15, 24} used in the `UniformlyDistributedDelay` process contained within each `EventGenerator`.

The output from each `EventSource` is output on its `outChannel` {16} to an `EventProcessing` process shown in Listing 9-11.

```
10 class EventProcessing implements CSProcess{
11
12   def ChannelInputList eventStreams
13   def minTime = 500
14   def maxTime = 750
15
16   void run() {
17     def mux2udd = Channel.one2one()
18     def udd2prn = Channel.one2one()
19     def pList = [
20                 new FairMultiplex ( inChannels: eventStreams,
21                                     outChannel: mux2udd.out() ),
22                 new UniformlyDistributedDelay ( inChannel:mux2udd.in(),
23                                                 outChannel: udd2prn.out(),
24                                                 minTime: minTime,
25                                                 maxTime: maxTime ),
26           new GPrint ( inChannel: udd2prn.in(),
27                        heading : "Event Output",
28                        delay: 0)
29           ]
30     new PAR (pList).run()
31   }
32 }
```

Listing 9-11 The EventProcessing Process Definition

The process `EventProcessing` receives inputs from each of the event streams on the `ChannelInputList eventStreams` {12}. The manner in which each event stream is selected is determined by the multiplexer {20–21}. The version shown in Listing 9-11 uses a `FairMultiplex` process, that ensures each stream is allocated an equal amount of the available output bandwidth. Other multiplexers can be used that have different properties, see `org.jcsp.groovy.util`. As in the single stream version the delay taken by processing the event is simulated by a `UniformlyDistributedDelay` process {22–25} before the events are printed using a `GPrint` process {26–28}.

The script that executes the multiStream version is shown in Listing 9-12.

```
10 def sources = Ask.Int ("Number of event sources between 1 and 9 ? ", 1, 9)
11
12 minTimes = [ 10, 20, 30, 40, 50, 10, 20, 30, 40 ]
13 maxTimes = [ 100, 150, 200, 50, 60, 30, 60, 100, 80 ]
14
15 def es2ep = Channel.one2oneArray(sources)
```

```
16
17 ChannelInputList eventsList = new ChannelInputList (es2ep)
18
19 def sourcesList = ( 0 ..< sources).collect { i ->
20            new EventSource ( source: i+1,
21            outChannel: es2ep[i].out(),
22            minTime: minTimes[i],
23            maxTime: maxTimes[i] )
24  }
15
26 def eventProcess = new EventProcessing ( eventStreams: eventsList,
27                                           minTime: 10,
28                                           maxTime: 400 )
29
30 new PAR( sourcesList + eventProcess).run()
```

Listing 9-12 The Run MultiStream Script

Initially the number of sources is determined by user interaction {10}. Then two lists {12, 13} are defined that specify the minimum and maximum time to be allocated to the `UniformlyDistributedDelay` process in each `EventSource`.

9.6 Summary

This chapter has shown that the adoption of parallel processing design techniques and implementation can shed a new light on age old computing problems. In particular, it allows designers to reason about both a system's behaviour and its performance when subjected to a large number of randomly occurring events.

9.7 Exercises

Exercise 9-1

Using the suggestion (Section 9.4.4) made earlier in the chapter, construct an additional process for the event handling system that ensures that the number of missed events is correct. The additional process should be added to the network of processes. You may need to modify the EventData class (Section 9.2.4) to facilitate this.

Exercise 9-2

The accompanying exercise package contains a version of the event handling system, `RunMultiStream`, which allows the creation of 1 to 9 event streams. By modifying the times associated with each event generation stream and also of the processing system explore the performance of the system. What do you conclude?

Exercise 9-3

The process `EventProcessing` has three versions of multiplexer defined within it, two of which are commented out. By choosing each of the options in turn, comment upon the effect that each multiplexer variation has on overall system performance.

Exercise 9-4

A manufacturing process utilises hoppers and a blender. The hoppers are used to hold raw materials and the blender is used to mix the contents from one or more hoppers. The collection of hoppers and the blender is managed by a controller. The hoppers indicate when they are ready to be used. The blender indicates when it is ready and also when mixing is to stop. The hoppers and the blender are clients to the server manager of the controller. The hoppers make a request to the manager to determine when they should stop processing raw materials. The aim of this exercise is to create three different control regimes as follows:

i. The hoppers and the blender indicate they are ready to start but mixing only commences when all three hoppers are ready after which the ready signal from the blender is ready.
ii. As in (i) above but mixing commences as soon as two hoppers and the blender are ready. If three hoppers are ready before the blender is ready then the last hopper is not used and will only be used during the following mixing cycle.

iii. As in (ii) above but mixing commences as soon as just one hopper and the blender are ready. If more than one hopper is ready before the blender becomes ready then these are retained until the following mixing cycle(s).

The accompanying exercises package has definitions for Hopper and Blender processes that utilise the GConsole to enable user interaction. These processes are complete and implement a client style behaviour. The user inputs an 'r' into the input area of the GConsole of the required Hopper or Blender to signify that it is ready. The Hopper or Blender process then outputs a '1' to signal to the Manager process that it is ready. The Manager then implements the required control regime as described above. A Hopper process then sends a '2' signal to the Manager indicating that it is ready to be stopped. The Blender process waits for the user to input an 'f' to indicate that blending can stop. The Blender then sends a '2' to the Manager process. The Manager then completes the client-server interactions. The web site contains scripts to execute each of the above control regimes. There are also outline process definitions for each of the control regimes that need to be completed. Initially, you are advised to produce a process network diagram to enable a better understanding of the interactions and process architecture.

10 Deadlock Revisited: Circular Structures

Some networks cannot be analysed using a client-server labelling

- a ring of processes invariably deadlocks
- plausible solutions are presented and then discounted
- deadlock avoidance strategies are described
- an argument is developed that shows that the final system will not deadlock

In previous chapters the concept of a client-server design pattern has been introduced and it has then been applied to a number of simple examples. The primary requirement of the pattern is that any resulting network should not contain any circuits of client and server labels. Needless to say, that if we have a ring of processes then a circuit is inevitable. Hence, we shall investigate a ring of processes to explore how, even though the client-server pattern cannot be applied, we can construct a system that is deadlock free.

The aim of the application is to construct a message passing structure from one node to another by providing a set of message passing elements that connect each node to the next. The simplest way of doing this is to create a ring of message passing nodes to which message sender and receiver processes are attached. Figure 10-1 shows the basic structure with a client-server labelling that demonstrates immediately that deadlock will occur, even ignoring the effect of the Sender and Receiver processes. It is obvious that the set of channels that connect the Ring Element processes has to be broken in some way. Deadlock will occur trivially when every Ring Element attempts to either input or output a message at the same time. Thus we have to find a way of breaking the ring

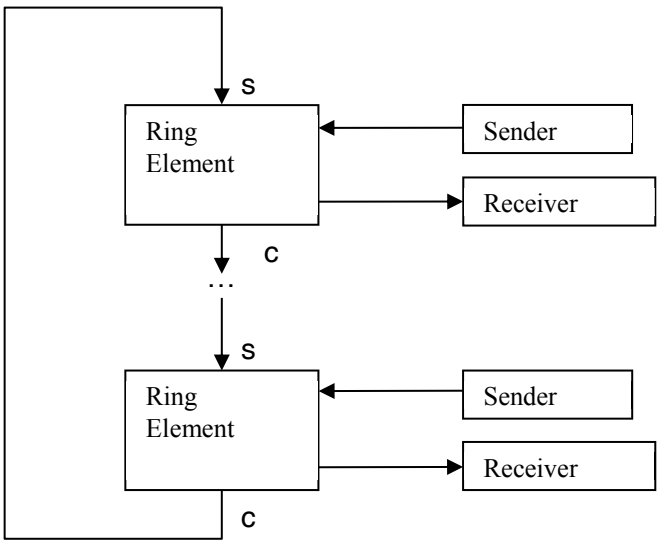


Figure 10-1 The Basic Message passing System

10.1 A First Sensible Attempt

The simplest way of the breaking the ring of channels connecting the Ring Element processes is to add another element to the ring which does the input and output operations in a different order to that undertaken by the Ring Elements. This will mean that there is at least one element on the ring that is always able to undertake an input operation if all the other Ring Elements are trying to output to the ring. This is shown in Figure 10-2.

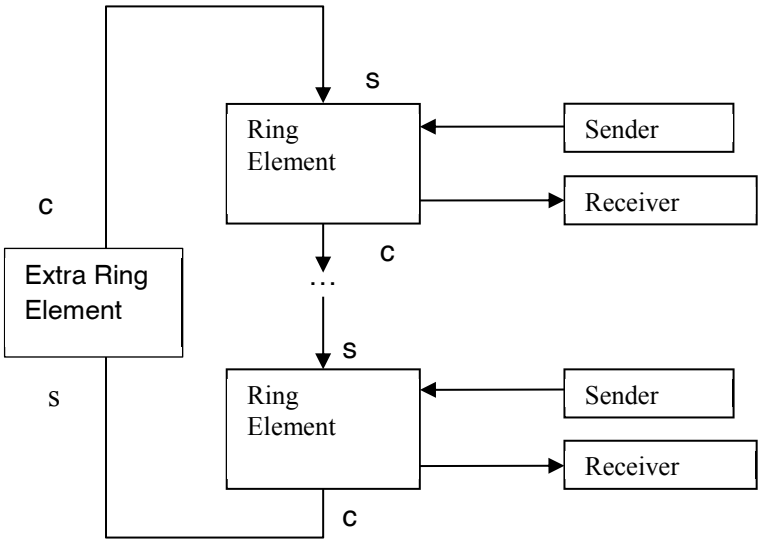


Figure 10-2 Adding the Extra Ring Element

The client-server labelling has not altered and still indicates a problem but we now know that the Extra Ring Element undertakes its input – output operations in a different order to the Ring Elements. The behaviour of a Ring Element is shown in Listing 10-1.

```
10 class RingElementv0 implements CSProcess {
11
12   def ChannelInput fromRing
13   def ChannelOutput toRing
14   def ChannelInput fromLocal
15   def ChannelOutput toLocal
16   def int element
17
18   void run () {
19     def RING = 0
20     def LOCAL= 1
21     def ringAlt = new ALT ( [ fromRing, fromLocal ] )
22     while (true) {
23       def index = ringAlt.priSelect()
24       switch (index) {
```

```
25          case RING:
26            def packet = (RingPacket) fromRing.read()
27            if ( packet.destination == element )
28              toLocal.write(packet)
29            else
30              toRing.write (packet)
31            break
32          case LOCAL:
33            def packet = (RingPacket) fromLocal.read()
34            toRing.write (packet)
35            break
36        }
37      }
38    }
39 }
```

Listing 10-1 The Ring Element Process Behaviour (Print Statements Omitted)

A Ring Element alternates over inputs from the ring and from its local sender process {21}. In a loop {22} it determines the enabled alternative, giving priority to inputs from the ring {23}. An enabled input from the ring is `read` {26} as a `RingPacket`, and if the message is for this element, it is written to the local receiver {28}, otherwise it is written to the ring {30} for onward transmission. If the enabled alternative is an input from the local sender then it is read {33} and written to the ring {34}.

The behaviour of the Extra Ring Element is shown in Listing 10-2.

```
10 class ExtraElement implements CSProcess {
11
12   def ChannelInput fromRing
13   def ChannelOutput toRing
14
15   void run () {
16     def packet = new RingPacket (source:-1, destination:-1,
17                                  value:-1, full: false )
18     while (true) {
19       toRing.write( packet )
20       packet = (RingPacket) fromRing.read()
21     }
22   }
23 }
```

Listing 10-2 The Behaviour of the Extra Ring Element (print Statements Omitted)

Given that the Ring Elements initially input from the ring, or local Sender process then the Extra Ring Element has to output a packet, so that a Ring Element has a packet to read. A `RingPacket` is defined {16, 17} which is then written to the ring {19}. Thereafter the process simply reads a `RingPacket` from the ring {20} and then outputs it to the ring {19}. The empty packet will continue to circulate forever.

10.1.1 Evaluation

The accompanying examples package contains a version of this first attempt `c10.examples.Runv0.groovy` that has print statements inserted within it to show the effect of this solution formulation. The user is able to indicate the number of nodes in the network when the system is executed. The messages received by each receiver process are displayed using a `GConsole` process. A network with 4 nodes will additionally have the extra node numbered as node 0. The output changes with each execution of the network but seldom terminates, though on occasion it has terminated. In a 4 node system each node should receive 3 messages from each of the other nodes, that is, each should receive 9 messages. Typically, no node receives all its messages and some nodes receive no messages. Inspection of the system console print messages indicates that the extra node does indeed output its empty packet and that this is read by the next node in the ring. This means that the other nodes have no input on their ring input channels and so they read a message from their local sender process. The sender processes attempt to send their messages as quickly as possible. This then has the effect of sending many messages on to the ring, which at some stage may deadlock when every node, including the extra node attempt to undertake an input or an output operation. Just when this occurs depends on the particular execution sequence. It is obvious that we have to find a way of managing the number of messages in the ring.

10.2 An Improvement

A simple improvement can be seen quite easily. If a node sends a message to another node on behalf of its local node then the receiving node undertakes to send a message back to the original source that the message has been read. This means that each node can only ever have one packet on the ring at any one time. On the first part of its journey it contains the desired message and then once it has been processed by the destination node it is returned with an empty flag. The definition of the `RingPacket` used to send messages around the system is shown in Listing 10-3. The property `source` {12} gives the number of the node that sent the message and `destination` {13} is the node to which it is to be sent. The actual message is contained in the property `value` {14} and the Boolean `full` {15} indicates whether the packet contains a message or is just an empty packet. A `toString` method is provided to enable printing of the packet on the console window and also on the `GConsole` processes.

```
10 class RingPacket implements Serializable, JCSPCopy {
11
12   def int source
13   def int destination
14   def int value
15   def boolean full
16
17   def copy () {
18     def p = new RingPacket ( source: this.source,
19                              destination: this.destination,
20                              value: this.value,
21                              full: this.full)
22     return p
23   }
24
25   def String toString () {
26     def s = "Packet [ s: ${source}, d: ${destination}, v: ${value}, f: ${full} ] "
27     return s
28   }
29 }
```

Listing 10-3 The RingPacket Class definition

A first running, `c10.examples.Runv1.groovy`, of this modification typically results in even worse performance than the initial version. On reflection this is obvious. The Extra Ring Element process still outputs an empty packet onto the ring and thus there will be no space for the messages to rotate around the ring. The solution is to modify the Extra Ring Element process so that it provides an empty space on the ring of nodes so that a communication can take place. This behaviour is shown in Listing 10-4. This does mean that the Extra Ring Element has to read {17} and then write {18} a packet, the same as all the other nodes.

```
10 class ExtraElementv1 implements CSProcess {
11
12   def ChannelInput fromRing
13   def ChannelOutput toRing
14
15   void run () {
16     while (true) {
17       def packet = (RingPacket) fromRing.read()
18       toRing.write( packet )
19     }
20   }
21 }
```

Listing 10-4 The Modified Behaviour of The Extra Ring Element (Print Statements Omitted)

The execution of `c10.examples.Runv1a.groovy` now results in the proper operation of the network with all Receivers getting and outputting the expected messages. The solution does however have some limitations in that only one packet is ever in circulation for each node as shown in the behaviour given in Listing 10-5.

The solution uses an alternative with pre-conditions to control the input of messages either from the ring or from the local sender {19–24}. Initially, messages can be input either from the ring or from the local sender {23, 24}. The `index` of the enabled alternative is determined using a `select` method call {26}.

For messages read from the ring {29}, it is first determined whether the message has its `destination` at this `element` {30}. It is then necessary to determine whether or not the packet is `full` {31}.

```
10    class RingElementv1 implements CSProcess {
11
12   def ChannelInput fromRing
13   def ChannelOutput toRing
14   def ChannelInput fromLocal
15   def ChannelOutput toLocal
16   def int element
17
18   void run () {
19     def RING = 0
20     def LOCAL= 1
21     def ringAlt = new ALT ( [ fromRing, fromLocal ] )
22     def preCon = new boolean[2]
23     preCon[RING] = true
24     preCon[LOCAL] = true
25     while (true) {
26       def index = ringAlt.select(preCon)
27       switch (index) {
28         case RING:
29           def packet = (RingPacket) fromRing.read()
30           if ( packet.destination == element ) {
31             if ( packet.full ) {
32               toLocal.write(packet.copy())
33               packet.destination = packet.source
34               packet.source = element
35               packet.full = false
36               toRing.write(packet)
37             }
38             else
39               preCon[LOCAL] = true
40           }
41           else
42             toRing.write (packet)
43           break
44         case LOCAL:
45           def packet = (RingPacket) fromLocal.read()
46           toRing.write (packet)
47           preCon[LOCAL] = false
48           break
49       }
50     }
51   }
52 }
```

Listing 10-5 The Ring Element That Expects A Returned Empty Packet (Print Statements Omitted)

If the packet is full then we can write a `copy` of the packet to the local receiver process {32}. After which we can update the content of the packet for its return journey to its originating node, because a copy was written to the local receiver process. The `destination` and `source` properties of the packet are updated accordingly {33, 34}, the `packet.full` indication is set `false` {35} and the revised packet written to the ring {36}. If the received packet is not `full` {38} then this is a returned packet and the ring element process can now input a message from its local sender, requiring an update to the associated pre-condition {39}.

If the initial packet was not destined for this node element {41} then it is simply written to the ring {42}.

Messages read from the local sender process {45} are immediately written to the ring {46} and the pre-condition controlling input from the local sender is set `false` {47}. As described above, this pre-condition will only be set `true`, when the returned empty packet has been received.

10.2.1 Evaluation

This solution, though functional, does still have some performance limitations in that an element has to wait for a sent packet to be returned before the next message can be sent. This means that on average half the network is filled with empty packets. The next solution removes this restriction by allowing the reuse of an empty packet if a node is ready to send a message from its local sender process.

10.3 A Final Resolution

The behaviour shown in Listing 10-6 shows the behaviour modification required to use an empty packet, as it passes through a node that is ready to output a local message.

```
10 class RingElementv2 implements CSProcess {
11
12   def ChannelInput fromRing
13   def ChannelOutput toRing
14   def ChannelInput fromLocal
15   def ChannelOutput toLocal
16   def int element
17
18   void run () {
19     def RING = 0
20     def LOCAL= 1
21     def ringAlt = new ALT ( [ fromRing, fromLocal ] )
22     def preCon = new boolean[2]
23     preCon[RING] = true
24     preCon[LOCAL] = true
25     def emptyPacket = new RingPacket ( source: -1, destination: -1 ,
26                                        value: -1 , full: false)
27     def localBuffer = new RingPacket()
```

```
28      def localBufferFull = false
29      toRing.write ( emptyPacket )
30      while (true) {
31        def index = ringAlt.select(preCon)
32        switch (index) {
33          case RING:
34            def ringBuffer = (RingPacket) fromRing.read()
35            if ( ringBuffer.destination == element ) {
36            toLocal.write(ringBuffer)
37            if ( localBufferFull ) {
38              toRing.write ( localBuffer )
39              preCon[LOCAL] = true
40              localBufferFull = false
41            }
42            else {
43              toRing.write ( emptyPacket )
44            }
45          }
46          else {
47            if ( ringBuffer.full ) {
48            toRing.write ( ringBuffer )
49            }
50            else {
51              if ( localBufferFull ) {
52              toRing.write ( localBuffer )
53              preCon[LOCAL] = true
54              localBufferFull = false
55            }
56            else {
57              toRing.write ( emptyPacket )
58            }
59          }
60        }
61        break
62      case LOCAL:
63        localBuffer = fromLocal.read()
64        preCon[LOCAL] = false
65        localBufferFull = true
66        break
67        } // end switch
68      }
69    }
70 }
```

Listing 10-6 The Final Ring Element Process (Print Statements Omitted)

The setup of the preconditions and the alternative are the same as the previous version {19–24}. An `emptyPacket` is defined {25, 26} as is a buffer {27} to hold messages from the local sender process. A Boolean flag, `localBufferFull` {28} is used to signify whether or not the `localBuffer` is full. The first action each `RingElement` node undertakes is to output an `emptyPacket` {29}, which has the effect of initialising the system. In general, this initial empty packet will only pass as far as the next node, which by then will have input a message from its local sender and will thus be able to use this empty packet. The extra element has the revised behaviour given in Listing 10-4. As before, the `index` of the enabled alternative is determined {31) and the appropriate `case` selected {32}.

If the `select`ed alternative is to read an input packet from the local sender process {62} this is read into the `localBuffer` {63}, the pre-condition flag for this alternative is set `false` {64} and the `localBufferFull` flag set `true`{65}. This does not cause the packet to be written to the ring, merely to get it ready to be written.

If the `select`ed alternative relates to an input from the ring then the message packet is read into a `ringBuffer` {34} and the subsequent processing is determined by the state of that message. If the `destination` of the message is for this node {35} then the message is written to the local receiver process {36}. This means that the node can output the `localBuffer` to the ring if it is full and update the flags associated with the buffer {37–40}; otherwise an `emptyPacket` is written to the ring {43}. If the `ringBuffer` does not have this node as its `destination` {46} then if the `ringBuffer` is full it is simply written to the ring {47–48}, otherwise the `localBuffer` is processed in the way described previously {51–58}.

10.3.1 Evaluation

This final version has resulted in a solution that routes messages around a circular network, which is inherently prone to deadlock. This version does not suffer from the drawbacks of the previous solution in that an empty packet only travels around the network until it comes to a node that needs to send a message from its `localBuffer` to another node. An argument has been presented that explains why deadlock will not occur because client-server labelling does not provide a categorical solution and furthermore indicates that deadlock will occur.

10.4 Summary

This chapter has analysed a set of processes that inherently tend to deadlock. Two algorithms have been developed that overcome the problems. The benefit of one solution over the other has been explained, though this is difficult to measure unless the system is run over a real network, where each Ring Element process can be placed on a specific processor of that network.

11 Graphical User Interfaces: Brownian Motion

This chapter introduces the concept of active user interface components

- active components are implemented by a process
- they have channel interfaces
- an alternative can be used to determine which component is ready for interaction
- the user interface has a declarative style with little or no coding for the interactions
- the application demonstrates how animation can be achieved
- the use of any2one and one2any channels is explained and justified

Previously, a simple user interface (`GConsole`) has been used that enables easier interpretation of the output from process networks. This chapter explores more complex user interfaces in conjunction with a relatively simple graphical application based upon particle movement.

The JCSP package contains an active implementation of the Java AWT (Abstract Windows Toolkit). The term active is here used to mean that each AWT component, for example, button, scrollbar and canvas, has been wrapped in a process so that component events and configuration are undertaken by channel communications. This means that the active components can be connected to any process. Furthermore, the programmer does not have to write any event handling or listener methods as these are contained within the active process wrapper. The active components inherit capabilities from the basic AWT components, thus the methods and fields associated with the component can be reused and active and ordinary, non-active, components can be used in the same interface.

The primary benefit of the active AWT components is that processes that access the user interface can utilise their non-deterministic capabilities, thereby reflecting the unpredictable behaviour of user interfaces. The user interface has no knowledge of when, for example, a button is going to be pressed and thus either a channel communication or an alternative provides a simple method for capturing that non-deterministic behaviour.

11.1 Active AWT Widgets

The fundamental process diagram for an active widget is shown in Figure 11-1. A widget is any component available in the `java.awt` package for which an active version has been constructed. Some active widgets have been constructed that simplify the construction of user interfaces. Specific widgets may have more or less channels depending upon the functionality of the widget.

All widgets have a configure input channel which enables the configuration of the widget at run-time. In most cases the configuration of a new widget can be defined when it is constructed, unless of course the content of the user interface is to be altered by changing the configuration of one of its widgets. For example, when a button has its associated text changed to reflect the state of the user interface. Each of the active component output channels produces data values that are related to the underlying AWT specification of that event and is specified in the `java.awt` documentation. The role of the configuration and event channels is specified in the `org.jcsp.awt` documentation and depends upon the specific component. For example, if the event arises from the pressing of an `ActiveButton` then the message communicated is the text string associated with the button. Similarly, a configuration channel message could be a text string that is to replace the current text associated with the button.

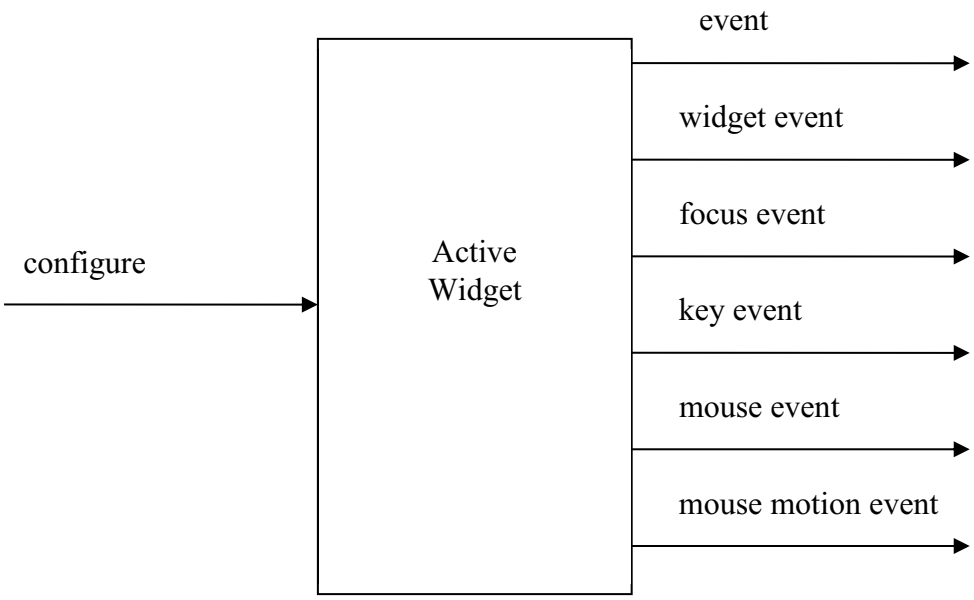


Figure 11-1 Generic Active Widget Process Diagram

11.2 The Particle System – Brownian Motion

A particle motion system (Lea, 2003) comprises a number of particles that move around at random. Their position is shown on a `Canvas`. Using Java `thread`s and a `Canvas` results in a somewhat cumbersome representation of the solution because a `Canvas` executes in its own thread of control, which has the effect of distributing the particle control, random movement and the graphical representation throughout the classes that make up the solution.

In the parallel solution that follows (see Figure 11-2) these drawbacks are eliminated and the fact that a `Canvas` has to execute in its own thread of control is hidden from the programmer. Furthermore in this solution we shall introduce some additional capabilities. The particles will bounce off the side of the bounding `Canvas`. The user will be given control of the application with a button that allows them to initially start the system and then subsequently to pause and resume its operation. In addition two buttons are provided which modify the 'temperature' of the system. The higher the temperature the greater the random movement exhibited by the particles. The particles do not bounce off each other and that is left as an additional exercise for the interested reader.

A number of particles (`Particle` 0 to n) are connected to the `ParticleInterface`. This utilises a new form of channel called `any2one`. An `any2one` channel enables the connection of any number of writer processes to a single reader process. The point-to-point nature of channel communication is, however, still maintained because only one communication can proceed at a time. Communications on an `any2one` are such that communication from one writer to the single reader is completed before the next writer can commence its communication. The converse is true of `one2any` channels. The JCSP library also includes `any2any` channels where yet again once a communication has started it behaves like a one-to-one communication.

`Particle`s are not aware of their position relative to the sides of the bounding `Canvas` and thus the particle may move to a position that is out with the bounding `Canvas`. In this case the particle's position is updated within the `ParticleInterface`. The updated position together with any change of temperature is returned to the `Particle` using the update channel. The update channel is a `one2any` channel that permits one writer to write to any number of readers. This is not a broadcast communication because the writer can only write to one of the reader processes at any one time. Furthermore, once one of the many reader processes has committed to a communication no other reader will be able to start a communication until the writer has written to that reader process.

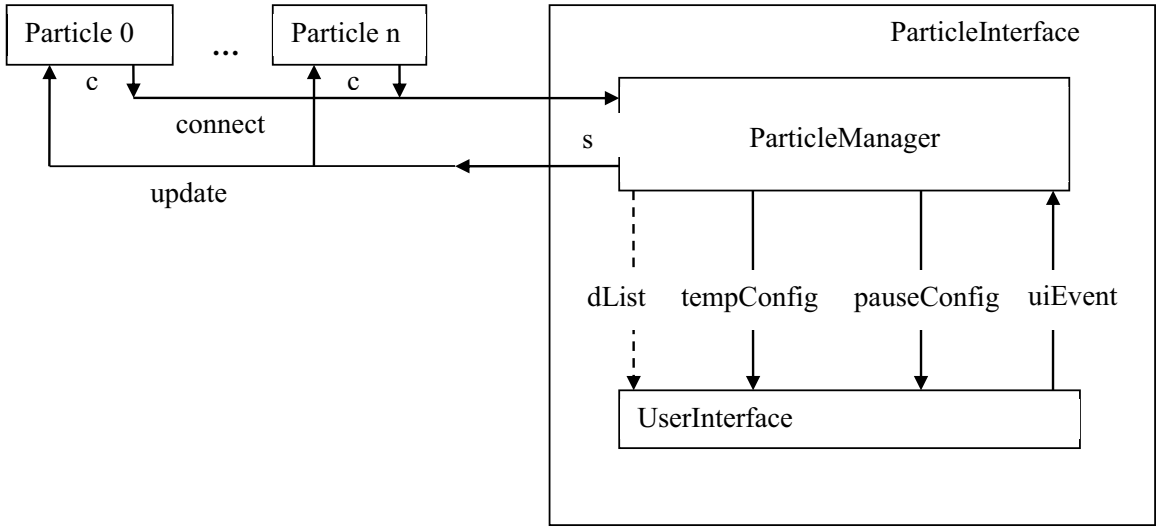


Figure 11-2 Brownian Motion Process Network

The `ParticleManager` is responsible for receiving inputs from the `Particle` processes; modifying their position, should the indicated position lie outside the bounding canvas; and then causing the display of the particle's position on the canvas. The `ParticleManager` is also responsible for dealing with button events from the `UserInterface` and configuring the buttons and labels within the `UserInterface`. Data is passed between the `Particle` processes and the `ParticleManager` by means of a data object that contains both positional information as well as any change to the temperature.

The `UserInterface` contains the display canvas, together with a button that is used to initially start and then subsequently used to pause and restart the system. Two further buttons are provided that are used to increase or decrease the temperature together with a `Label` that shows the current temperature value with an indication of whether the last change was up or down. The channels used between the `ParticleManager` and the `UserInterface` will be described more fully in a later section.

11.2.1 The Position Data Object

The `Position` data object, see Listing 11-1, is used to communicate data between the Particles and the ParticleInterface.

`Position` implements the interface `JCSPCopy` {10}, which is defined within the `org.jcsp.groovy` package. It should be recalled that objects are passed between processes running on the same machine by means of an object reference. In some situations this could lead to the creation of a large number of newly created short-lived objects, which could lead to the calling of the automatic Java garbage collector. The calling of the garbage collector during a graphical display would interfere with the presentation. The abstract interface `JCSPCopy` defines a method called `copy()`, which can be used to generate a deep copy of an object.

Lines {12–17} define the properties of `Position`. The property `id` is the number of the `Particle`. The properties `lx` and `ly` are the newly calculated [x, y] position co-ordinates of the `Particle`. These co-ordinates may lie outside the display area. The properties `px` and `py` are the co-ordinates of the previous position of the particle. The property `temperature` maintains the current value of the temperature within the system. All the properties, apart from `id` can be altered within the `ParticleInterface`.

```
10 class Position implements JCSPCopy {
11
12   def int id // particle number
13   def int lx // current x location of particle
14   def int ly // current y location of particle
15   def int px // previous x location of particle
16   def int py // previous y location of particle
17   def int temperature // current working temperature
18
19   def copy() {
20     def p = new Position ( id: this.id,
21                            lx: this.lx, ly: this.ly,
22                            px: this.px, py: this.py,
23                            temperature : this.temperature )
24     return p
25   }
26
27   def String toString () {
28     def s = "[Position-> " + id + ", " + lx + ", " + ly
29     s = s + ", " + px + ", " + py
30     s = s + ", " + temperature + " ]"
31     return s
32   }
33 }
```

Listing 11-1 The Position Data Object

Lines {19–25} define the method `copy` required for the implementation of the interface `JCSPCopy`. For completeness, a `toString` method is defined {27–32} that can be used to output the contents of a `Position` object.

11.2.2 The Particle Process

The definition of the `Particle` process is shown in Listing 11-2.

```
10 class Particle implements CSProcess {
11
12   def ChannelOutput sendPosition
13   def ChannelInput getPosition
14   def int x = 100 // initial x location
15   def int y = 100 // initial y location
16   def long delay = 200 // delay between movements
17   def int id
18   def int temperature = 25 // in range 10 to 50
19
20   void run() {
21     def timer = new CSTimer()
22     def rng = new Random()
23     def p = new Position ( id: id, px: x, py: y, temperature: temperature )
24     while (true) {
25       p.lx = p.px + rng.nextInt(p.temperature) - ( p.temperature / 2 )
```

```
26        p.ly = p.py + rng.nextInt(p.temperature) - ( p.temperature / 2 )
27        sendPosition.write ( p )
28        p = ( (Position)getPosition.read() ).copy()
29        timer.sleep ( delay )
30      }
31    }
32 }
```

Listing 11-2 The Particle Process

A `Particle` has two channels one {12}, `sendPosition`, to output its `Position` to, and the other {13}, `getPosition`, to receive updated `Positions` from the `ParticleInterface`. It should be noted that even though these channels will eventually be implemented as `any2one` and `one2any` channels as far as the process is concerned these are just a `ChannelOutput` and `ChannelInput` respectively. The properties `x` {14} and `y` {15} hold the initial position of the particle. A default display area of 200 pixels is presumed and thus all particles start their movement from the centre of that area. The position of the particles will be recalculated after the interval specified by `delay` {16}, which is initially set to 200 milliseconds. Each `Particle` is given a unique identification `id` {17}. The initial `temperature` of the system is set at 25 {18} and can range from 10 to 50.

The `run` method defines a `CSTimer` called `timer` {21} and uses the Java provided random number generator mechanism, `Random ()` {22}. The variable `p` holds the `Position` of the particle and is constructed using the initial values held within the properties passed to the process {23}.

The main loop of the process {24-30} requires the calculation of the new position of the particle `lx` and `ly` that are stored in the variable object p {25, 26}. The calculation ensures that the particle moves in a space that surrounds the current location [`px`, `py`] by a square with a side of size `temperature`. The position `p` is then written to the `ParticleInterface` {27}. This is a `write` operation that is implemented on a shared `any2one` channel and thus the process will have to wait until any other outstanding communications have completed. An `any2one` channel is essentially fair in that the communications are placed in a queue of such communications.

The `Particle` process behaves like a client to the `ParticleInterface`'s server. As soon as it has written its position to the `ParticleInterface` it `read`s the updated position information {28} from the `getPosition` channel. The `getPosition` channel is implemented by means of a `one2any` channel and thus this client – server interaction has to be carefully considered. When the `sendPosition.write(p)` {27} communication is completed only this `Particle` process can be in that state because only one communication is permitted on an `any2one` channel. Hence the only process that will be in a position to undertake a `read` on the `getPosition` channel is this process. Hence we are assured that a `Particle` process that writes it's position to `ParticleInterface` will be the one to receive its response, even though we are using shared `any2one` and `one2any` channels.

Finally, the Particle process sleeps for the delay period {29} after which the loop is repeated until the user stops the application through the user interface. The user interface will cause the Particle processes to stop even though they are implemented using a non-terminating while-loop.

11.2.3 The Particle Interface Process

This process, shown in Listing 11-3 is typical of any application that uses a graphical user interface in that it comprises a process that undertakes both the interaction with the user interface and the rest of the system and the process that implements the user interface itself. These two processes are always run in parallel using communication channels to pass events and configuration information between the processes.

The channels `inChannel` {12} and `outChannel` {13} are used to connect this process to the Particle processes. Yet again this process definition does not need to be aware of the specific implementation of the channels actually used to connect the processes together. The property `canvasSize` {14} provides a default size for the display area. Similarly, properties are defined for the number of `particles` {15}, the `centre` of the display area {16} and the `intialtemp`(erature) {17} of the system.

The variable `dList` {20} is of type `DisplayList`, defined within `org.jcsp.awt`. The use of `dList` will be described later. It is sufficient to note, at this stage, that it is passed as a property to the `ParticleManager` process {28}. An `ActiveCanvas`, `particleCanvas` is defined {21} and then a call to its `setPaintable()` method is made that associates it with `dList` {22}. In this manner both `ParticleManager` and `UserInterface` can access `dList`, the former directly as a property and the other indirectly through `particleCanvas` {36}. Essentially, `dList` is a shared object between the processes but the user can only modify the `dList` in `ParticleManager` directly. Therefore a `DisplayList` object has to be defined before either of the processes that access it are defined. A DisplayList is the mechanism by which animation can be more easily achieved.

```
10 class ParticleInterface implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14   def int canvasSize = 100
15   def int particles
16   def int centre
17   def int initialTemp
18
19   void run() {
20     def dList = new DisplayList()
21     def particleCanvas = new ActiveCanvas()
22     particleCanvas.setPaintable (dList)
23     def tempConfig = Channel.one2one()
```

```
24      def pauseConfig = Channel.one2one()
25      def uiEvents = Channel.any2one( new OverWriteOldestBuffer(5) )
26      def network = [ new ParticleManager ( fromParticles: inChannel,
27                                            toParticles: outChannel,
28                                            toUI: dList,
29                                            fromUIButtons: uiEvents.in(),
30                                            toUIPause: pauseConfig.out(),
31                                            toUILabel: tempConfig.out(),
32                                            CANVASSIZE: canvasSize,
33                                            PARTICLES: particles,
34                                            CENTRE: centre,
35                                            START_TEMP: initialTemp ),
36                      new UserInterface ( particleCanvas: particleCanvas,
37                                          canvasSize: canvasSize,
38                                          tempValueConfig: tempConfig.in(),
39                                          pauseButtonConfig: pauseConfig.in(),
40                                          buttonEvent: uiEvents.out() )
41                      ]
42      new PAR ( network ).run()
43    }
44 }
```

Listing 11-3 The ParticleInterface Process

The `tempConfig` channel {23} is used to update the temperature display in the interface. The `pauseConfig` {24} channel is used to set the text in the `START/PAUSE/RESTART` button.

The `uiEvents` channel {25} passes button events from the `UserInterface` to the `ParticleManager` process. It is not possible to press two buttons at the same time hence we can use an `any2one` channel, which simplifies processing within the `ParticleManager` process. The parameter `OverWriteOldestBuffer (5)` specifies that this channel will use a buffer of 5 elements in which, should it become full the oldest element in the buffer will be overwritten. This buffer is required because it is essential that events on this channel are always read otherwise the underlying Java event thread may block, which would also have the effect of stopping the rest of the user interface. The specified buffer will always read an input, hence ensuring that the Java event thread will not block and that another process will always be able to read the last few events, five in this case, even if the reading process is slow.

The `network` {26–41} simply comprises the `ParticleManager` and `UserInterface` processes with parameters and variables passed as parameters as required to construct the process network as shown in Figure 11-2.

11.2.4 The ParticleManager Process

The properties of the `ParticleManager` process are shown in Listing 11-4. The channel connections with `Particle` processes are provided by the channels `fromParticles` {12} and `toParticles` {13}. When the system is instantiated these will be passed shared channels of type `any2one` and `one2any` respectively. The constant properties {15–18} respectively contain the size of the square display area (`CANVASSIZE`), number of particles (`PARTICLES`), the centre co-ordinate of the display area (`CENTRE`) and the initial value of the system temperature (`START_TEMP`). The `DisplayList` property {14}, `toUI`, provides the graphical connection between the `ParticleManager` and `UserInterface` processes. The `ChannelInput` {19}, `fromUIButtons`, is the channel by which button events from the user interface are communicated to `ParticleManager`. Finally, the `ChannelOutputs` `toUILabel` {20} and `toUIPause` {21} provide the means by which the temperature value and the `START`, `PAUSE` and `RESTART` button have their values changed.

```
10 class ParticleManager implements CSProcess {
11
12   def ChannelInput fromParticles
13   def ChannelOutput toParticles
14   def DisplayList toUI
15   def int CANVASSIZE
16   def int PARTICLES
17   def int CENTRE
18   def int START_TEMP
19   def ChannelInput fromUIButtons
20   def ChannelOutput toUILabel
21   def ChannelOutput toUIPause
22
```

Listing 11-4 ParticleManager Properties

The initialisation of the `ParticleManager` is shown in Listing 11-5. The variable `colourList` {24–26} contains a list of `java.awt.colors` that is used to colour the particles once they start moving. The variable `temperature` {28} is assigned the value of property `START_TEMP`.

The next part {30–46} initialises the variables that will be used by the `DisplayList` mechanism. The variable, `particleGraphics` {30}, used to `set()` a `DisplayList` comprises an array of `GraphicsCommands`. The initial element of `particleGraphics` {32} contains a `GraphicsCommand` that clears the display area. The remainder of `particleGraphics` comprises two elements per particle. The first element of which is a command to set the colour of the particle and the second will draw a circle of that colour with a radius of 10 pixels at the position of the particle. However for initialisation, each particle is set to the colour `BLACK` {36} and placed at the `CENTRE` {37} of the display area. This is captured in the variable `initialGraphic` {34}. The nested for loops {39–44} copy the `initialGraphic` into the array `particleGraphics`. Thus `particleGraphics` comprises a first command to clear the display followed by as many pairs of `GraphicsCommands` as there are particles needing to be drawn. The `DisplayList`, `toUI` is then `set()` to `particleGraphics` {46}. The manner in which the `DisplayList` is manipulated will be described later.

The two element array `positionGraphic` {47–51} will subsequently be used to update the `DisplayList` to reflect the movement of particles. It is initialised to sensible values that will be overwritten. However it can be observed that the first element of the array contains a command to set the colour and the second causes the drawing of a circle of that colour. The `ParticleManager` process alternates over inputs from the user interface buttons, `fromUIButtons` and from the particles on channel `fromParticles` {53}. The `String initTemp` is defined to hold the initial value of `temperature` {55} surrounded by spaces. This `String` is then written to the label that displays this value using the channel `toUILabel` {56}.

```
23    void run() {
24      def colourList = [ Color.BLUE, Color.GREEN,
25                         Color.RED, Color.MAGENTA,
26                         Color.CYAN, Color.YELLOW]
27
28      def temperature = START_TEMP
29
30      GraphicsCommand[] particleGraphics = new GraphicsCommand[1+(PARTICLES*2)]
31
32      particleGraphics[0] = new GraphicsCommand.ClearRect(0, 0,
        CANVASSIZE,CANVASSIZE)
33
34      GraphicsCommand [] initialGraphic = new GraphicsCommand [ 2 ]
35
36      initialGraphic[0] = new GraphicsCommand.SetColor (Color.BLACK)
37      initialGraphic[1] = new GraphicsCommand.FillOval (CENTRE, CENTRE, 10, 10)
38
```

```
39      for ( i in 0 ..< PARTICLES ) {
40        def p = (i * 2) + 1
41        for ( j in 0 ..< 2) {
42          particleGraphics [p+j] = initialGraphic[j]
43        }
44      }
45
46      toUI.set (particleGraphics)
47      GraphicsCommand [] positionGraphic = new GraphicsCommand [ 2 ]
48      positionGraphic =
49        [ new GraphicsCommand.SetColor (Color.WHITE),
50          new GraphicsCommand.FillOval (CENTRE, CENTRE, 10, 10)
51        ]
52
53      def pmAlt = new ALT ( [fromUIButtons, fromParticles] )
54
55      def initTemp = " " + temperature + " "
56      toUILabel.write ( initTemp )
57
58      def direction = fromUIButtons.read()
59      while ( direction != "START" ) {
60        direction = fromUIButtons.read()
61      }
62      toUIPause.write("PAUSE")
63
```

Listing 11-5 ParticleManager Initialisation

The variable `direction` is read from the channel `fromUIButtons` {58}. A user interface button signals a button event by communicating the `String` that is currently displayed by the button. Recall that all the user interface buttons are connected to the same channel, `fromUIButtons`. Only the `START`/`PAUSE`/`RESTART` button has the initial value `START` and thus the process will wait until the button labelled `START` is pressed. This behaviour is captured in the `while` loop {59–61}, which ignores any other button events. Once `START` has been read, the button's text value is changed to `PAUSE` {62} by writing to the `toUIPause` channel. The operation of the system now commences and this is shown in Listing 11-6.

The `index` of the selected alternative is obtained, with priority being given to button events {65}. If the value `read` from the channel `fromUIButtons` is `PAUSE` {68} then it is immediately overwritten with `RESTART` {69}. The process then waits for the button event `RESTART` ignoring all other button events {71–73}. Once the system has been restarted the button is overwritten with the value `PAUSE` {74}.

```
64      while (true) {
65        def index = pmAlt.priSelect()
66        if ( index == 0 ) { // dealing with a button event
67          direction = fromUIButtons.read()
68          if (direction == "PAUSE" ) {
69            toUIPause.write("RESTART")
70            direction = fromUIButtons.read()
71            while ( direction != "RESTART" ) {
72              direction = fromUIButtons.read()
73            }
74            toUIPause.write("PAUSE")
75          }
76          else {
77            if (( direction == "Up" ) && ( temperature < 50 )) {
78                 temperature = temperature + 5
79                 def s = "+" + temperature + "+"
80                 toUILabel.write ( s )
81            }
82            else {
83              if ( (direction == "Down" ) && ( temperature > 10 ) ) {
84                 temperature = temperature - 5
85                 def s = "-" + temperature + "-"
86                 toUILabel.write ( s )
```

```
87                }
88                else {
89                }
90              }
91            }
92          }
```

Listing 11-6 ParticleManager Button Event Processing

If the value `read` into `direction` is not `PAUSE` then it must either be `Up` or `Down` which are the text strings associated with the buttons that manipulate the `temperature` of the system. If the `Up` button is pressed and provided the current value of `temperature` is less than 50 {70} then the `temperature` is raised by 5 {78} and the new value of `temperature` is written to the interface using the channel `toUILabel` surrounded by + symbols {79–80}. Similarly if the `Down` button is pressed then the `temperature` is reduced by 5 provided its current value is greater than 10 and is output surrounded by – symbols {83–89}.

Listing 11-7 shows the processing that deals with the movement of particles.

```
93        else { // index is 1 particle movement
94          def p = (Position) fromParticles.read()
95          if ( p.lx > CANVASSIZE ) { p.lx = (2 * CANVASSIZE) - p.lx }
96          if ( p.ly > CANVASSIZE ) { p.ly = (2 * CANVASSIZE) - p.ly }
97          if ( p.lx < 0 ) { p.lx = 0 - p.lx }
98          if ( p.ly < 0 ) { p.ly = 0 - p.ly }
99          positionGraphic [0] = new GraphicsCommand.SetColor( colourList.
            getAt(p.id%6))
100         positionGraphic [1] = new GraphicsCommand.FillOval (p.lx, p.ly, 10, 10)
101         toUI.change ( positionGraphic, 1 + ( p.id * 2) )
102         p.px = p.lx
103         p.py = p.ly
104         p.temperature = temperature
105         toParticles.write(p)
106       } // index test
107     } // while
108   } // run
109 }
```

Listing 11-7 ParticleManager Particle Movement Processing

Recall that ParticleManager is behaving as a server process. Hence we would expect to see it read a client request {94}, undertake some processing and then respond with the return value {105}. The `Position` data object is `read` into the variable `p` from the channel `fromParticles` {94}. The proposed location `[lx, ly]` of the particle is then assessed as to whether it still remains within the display area {95–98} and if not, its position is adjusted assuming that the reflection from the side of the display area involves no friction or elastic compression of the particle. The value of the `PositionGraphic` array is then modified to reflect the particle's colour by taking the modulus 6 remainder of the particle's id {99} and then setting the centre of the circle to `[lx, ly]` {100}. This is then used to overwrite the data for this particle in the DisplayList parameter using the `toUI.change()` method {101}.

The position of the particle can now be updated {102, 103}. The current value of temperature is assigned to the corresponding property of object `p` {104} and the updated object `p` is then written back to the waiting `Particle` process {105}, as described in Section 11.2.2.

The description of the operation of a `DisplayList` can now be completed. An `ActiveCanvas` takes the `DisplayList` object as a parameter. Internally, the ActiveCanvas constructs two copies of the associated DisplayList array of Graphics commands. These copies are used to provide a double buffering mechanism; this however is hidden from the programmer. At a specified period the `ActiveCanvas` draws the current buffer on the display, while other changes are recorded in the other copy. This mechanism is repeated displaying the first buffer and recording changes in the second and then displaying the second buffer while recording changes in the first copy.

The `DisplayList` is initialised by a set method {46}. Thereafter specific elements of the `DisplayList` can be altered using the change method {101}. Thus the programmer generates the effect of continually updating the display, which in fact is using a double buffering technique to smooth the repainting of the display. The user is not concerned with the repainting of the display as this handled within the `ActiveCanvas` process. Thus the `DisplayList` array of `GrahicsCommand`s has an initial element that clears the display area, which is then overwritten by the sequence of `GraphicsCommands` in the array. In this manner sophisticated animation can be achieved, without having to overwrite each particle individually.

11.2.5 The UserInterface Process

The `UserInterface` process is shown in Listing 11-8. The properties of the process include the `particleCanvas`, `fromPM` {12}, the size of the canvas {13}, the two input channels, `tempValueConfig` {14} and `pauseButtonConfig` {15} used to configure the temperature value and the start button. Finally, the `buttonEvent` channel is used to output button events to the `ParticleManager` process {16}.

The `run` method of this process comprises a declarative style list of definitions and associated method calls that instantiates the graphical user interface. First, `root` {19}, an `ActiveClosingFrame` is defined that will be used to hold the rest of the interface components. An `ActiveClosiongFrame` is defined with the frame's title as a parameter and is not introduced by a property name because these processes are defined as Java classes and thus are constructed using the normal Java mechanism. `ActiveClosingFrame` is a specialisation of `ActiveFrame` that permits the closing of the frame using the normal window based controls. Interface components have to be added to the enclosed frame which is accessed by means of the `getActiveFrameMethod()` call {20}. The next part of the Listing shows the definition of the interface widgets both active and ordinary AWT non-active ones which can be mixed as required. The `Label`, `tempLabel`, which displays the text '`Temperature`' is constructed {21}. An `ActiveLabel` called `tempValue` is then defined {22} with the channel `tempValueConfig` as its parameter. Typically, an active widget has a constructor that comprises the configuration and event channels, together with any other appropriate parameter. The alignment of the label is also specified {23}. After this the required `ActiveButton`s are defined {24–26}, in which the `null` parameter is a placeholder for the not needed configuration channel. The additional parameter specifies the initial text associated with the button. The `pauseButton` requires a configuration channel {26} because the value of the text `String` associated with the button changes as the application progresses.

```
10 class UserInterface implements CSProcess {
11
12   def ActiveCanvas particleCanvas
13   def int canvasSize
14   def ChannelInput tempValueConfig
15   def ChannelInput pauseButtonConfig
16   def ChannelOutput buttonEvent
17
18   void run() {
19     def root = new ActiveClosingFrame ("Brownian Motion Particle System")
20     def mainFrame = root.getActiveFrame()
21     def tempLabel = new Label ("Temperature")
22     def tempValue = new ActiveLabel (tempValueConfig)
23     tempValue.setAlignment( Label.CENTER)
24     def upButton = new ActiveButton (null, buttonEvent, "Up")
25     def downButton = new ActiveButton (null, buttonEvent, "Down")
26     def pauseButton = new ActiveButton(pauseButtonConfig, buttonEvent, "START" )
27     def tempContainer = new Container()
28     tempContainer.setLayout ( new GridLayout ( 1, 5 ) )
29     tempContainer.add ( pauseButton )
30     tempContainer.add ( tempLabel )
31     tempContainer.add ( upButton )
32     tempContainer.add ( tempValue )
33     tempContainer.add ( downButton )
34     particleCanvas.setSize (canvasSize, canvasSize)
35     mainFrame.setLayout( new BorderLayout() )
36     mainFrame.add (particleCanvas, BorderLayout.CENTER)
37     mainFrame.add (tempContainer, BorderLayout.SOUTH)
38     mainFrame.pack()
39     mainFrame.setVisible ( true )
40     def network = [root, particleCanvas, tempValue, upButton,
       downButton, pauseButton]
41     new PAR (network).run()
42   }
43 }
```

Listing 11-8 The User Interface Process

Next a `Container`, `tempContainer` is defined {27} that holds all the components associated with the manipulation of temperature together with the `pauseButton`. The `Container` uses a `GridLayout` {28}. The previously defined buttons and labels are then added to `the tempContainer` {29–33}. The size of `particleCanvas` is specified {34}.

The `mainframe` can now be created {35–39} by specifying it to be a `BorderLayout` {35}. The `particleCanvas` and `tempContainer` are then `added` to the `mainframe` in the `CENTER` and `SOUTH` of the layout {36, 37}. The `mainframe` is then `packed` and `setVisible` {38, 39}, in the manner normally required by AWT interfaces.

Finally, a process `network` is constructed that comprises the root and the remaining active widgets {40}. The `network` is then run {41} and that is all that needs to be specified for the user interface requirements of this application. The event handler and listener methods normally required do not have to be written as these have been encapsulated within the active widgets, thereby simplifying the construction of the user interface.

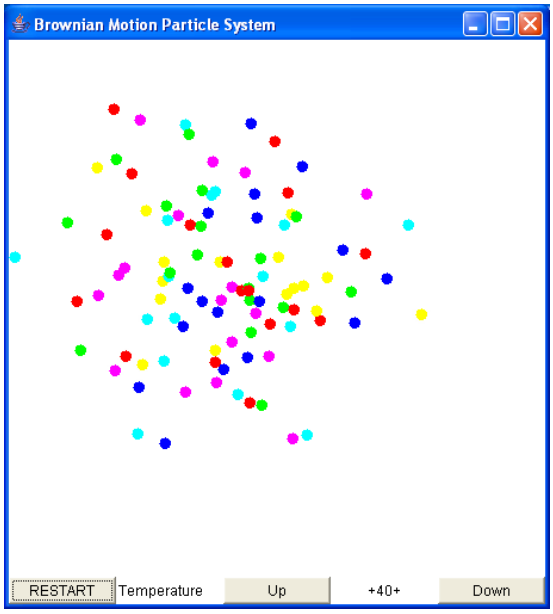
11.2.6 Invoking the Brownian Motion System

Listing 11-9 gives the script that is required to invoke the Brownian motion system. The `any2one` channel `connect` and the `one2any` channel `update` are defined {10, 11}. The fundamental constants of the system are either obtained from a user interaction or defined as constants {13-16}. The empty `List network` is defined {18} to which is appended each of the `Particle` processes {20–26}. The `ParticleInterface` process is finally appended to `network` {28–33}. The system is then executed by `run`ning `PAR` {35}.

```
10 def connect = Channel.any2one()
11 def update = Channel.one2any()
12
13 def CSIZE = Ask.Int ("Size of Canvas (200, 600)?: ", 200, 600)
14 def CENTRE = CSIZE / 2
15 def PARTICLES = Ask.Int ("Number of Particles (10, 200)?: ", 10, 200)
16 def INIT_TEMP = 20
17
18 def network = []
19 for ( i in 0..< PARTICLES ) {
20   network << new Particle ( id: i,
21                             sendPosition: connect.out(),
22                             getPosition: update.in(),
23                             x: CENTRE,
24                             y: CENTRE,
25                             temperature: INIT_TEMP )
26 }
27
28 network << ( new ParticleInterface ( inChannel: connect.in(),
29                                      outChannel: update.out(),
30                                      canvasSize: CSIZE,
31                                      particles: PARTICLES,
32                                      centre: CENTRE,
33                                      initialTemp: INIT_TEMP ) )
34 println "Starting Particle System"
35 new PAR ( network ).run()
```

Listing 11-9 The Script To Invoke the Brownian Motion System

A typical screen capture of the system, when it has been PAUSEd is shown in Output 11-1. We can observe that the control button has been set to RESTART. The temperature is currently set at 40 and the last operation was to increase its value because it is surrounded by + symbols. The Up and Down buttons are clearly visible. The screen is derived from a system that has a canvas size of 450 pixels running 100 particles.



Output 11-1 Screen Capture of the Brownian Motion System

11.3 Summary

This chapter has described how user interfaces can be constructed very simply using the active widget concept. Of most significance is the relative simplicity of the user interface definition as it does not require the programmer to implement the event and listener methods normally required. It has introduced a standard design pattern for user interface applications in which there is a process that undertakes the processing `ParticleManager` and its associated `UserInterface` process that are executed in parallel.

The concept of a `DisplayList` has been introduced which simplifies the programming of animated user interfaces based upon drawing in an `ActiveCanvas`. This in itself typifies the ease with which user interfaces can be constructed using active widgets because the programmer can use the parallel programming constructs to implement the interaction between user and application processes.

The design and implementation of user interfaces has become a much easier task because the user is no longer concerned with the writing of event handler and listener methods. Furthermore, the encapsulation of interface components, which run in their own thread and their associated event handler thread into a single process, makes it much easier to build the system that interacts with the interface.

11.4 Exercises

Exercise 111

The Control process in the Scaling system (Chapter 5) currently updates the scaling factor according to an automatic system. Replace this with a user interface that issues the suspend communication, obtains the current scaling factor and then asks the user for the new scaling factor that is then injected into the `Scaler`. The original and scaled values should also be output to the user interface. There is a widget called ActiveTextEnterField that may be useful (see the JCSP documentation).

12 Dining Philosophers: A Classic Problem

Dining Philosophers is one of the best known and most used of the classic problems in concurrent and parallel programming. This chapter:

- defines the problem
- explains its continuing relevance
- develops two different solutions, each of which begins with a faulty design

This problem first formulated by Dijkstra is cited by Hoare in his original paper on Communicating Sequential Processes (Hoare, 1978). Tantalisingly, Hoare presents the problem and a partial solution leaving it up to the reader to finish the solution. The problem was formulated at a time, in the mid-1970s, when computer manufacturers were having a great deal of difficulty in building operating systems that were correct and could withstand continued use. Typical problems that had to be overcome were deadlock between different tasks and other tasks being starved of resources; exactly the same problems that the client-server design pattern solves.

The problem has the following statement. Five philosophers spend their lives thinking and eating. They share a common dining room in their college where there is a circular table surrounded by five chairs, each is assigned to one of the philosophers. In the centre of the table there is a large bowl of spaghetti. The table is set with five forks each one assigned to a specific philosopher. On feeling hungry the philosopher enters the room, sits in his own chair and picks up his fork, which is to his left hand. The spaghetti is so tangled that he needs to use the fork to his right hand side as well. When he has finished eating he replaces both forks and leaves the room. The college has provided a butler who ensures that the bowl of spaghetti is always full and can carry out other duties as necessary such as washing-up and guiding philosophers to their own seat.

It is apparent that the critical aspect of this problem is in the management of the forks. If a philosopher is never able to pick up the fork to their right then they will never be able to eat and will thus exhibit starvation or as we have termed it, livelock. Similarly, if all the philosophers enter the room at the same time and each picks up their own left fork none of them will be able to pick up their neighbour's fork to their right and thus deadlock will ensue as none of the philosophers will ever be able to eat.

12.1 Naïve Management

The behaviour of a philosopher is relatively simple and is captured in Listing 12-1. A `Philosopher` can access their own `leftFork` {12}, and their neighbour's as their `rightFork` {13} they can also `enter` {14} into or `exit` {15} from the room. A set of output channels is provided for each Philosopher so they can indicate their intentions. A philosopher is identified by a property `id` {16}. The behaviour of each philosopher will be governed by a `timer` {18}.

A method, `action`, has been provided {20-23} that prints the current action of a philosopher and also makes them wait for a specified period. A Philosopher is initially thinking for 1 second {27}, after which they enter the room {28}. They then indicate they are picking up their left fork by means of a signal {30} and similarly for their right fork {32}.

They are then eating for 2 seconds {34}, after which they put down their left fork {35}, then their right fork {37} and then they leave the room {39} to resume thinking {27}. After each successful interaction an appropriate message is printed to the console. The messages from the Philosophers will be interleaved on the console but in this case that is precisely what is required as we want to see how the Philosophers interact with each other.

```
10 class Philosopher implements CSProcess {
11
12   def ChannelOutput leftFork
13   def ChannelOutput rightFork
14   def ChannelOutput enter
15   def ChannelOutput exit
16   def int id
17
18   def timer = new CSTimer()
19
20   def void action ( id, type, delay ) {
21     println "${type} : ${id} "
22     timer.sleep(delay)
23   }
24
25   void run() {
26     while (true) {
27       action (id, " thinking", 1000 )
28       enter.write(1)
29       println "$id: entered"
30       leftFork.write(1)
31       println "$id: got left fork"
32       rightFork.write(1)
33       println "$id: got right fork"
34       action (id, " eating", 2000 )
35       leftFork.write(1)
36       println "$id: put down left"
37       rightFork.write(1)
38       println "$id: put down right"
39       exit.write(1)
40       println "$id: exited"
41     }
42   }
43 }
```

Listing 12-1The Behaviour of a Philosopher

A Fork, Listing 12-2, can either be picked up from the right or the left depending upon which Philosopher has sat down. These are indicated by a signal on the appropriate channel, `left` {12}, or `right` {13}.

```
10 class Fork implements CSProcess {
11
12   def ChannelInput left
13   def ChannelInput right
14
15   void run () {
16     def fromPhilosopher = [left, right]
```

```
17      def forkAlt = new ALT ( fromPhilosopher )
18      while (true) {
19        def i = forkAlt.select()
20        fromPhilosopher[i].read() //pick up fork i
21        fromPhilosopher[i].read() //put down fork i
22      }
23    }
24 }
```

Listing 12-2 The Fork Behaviour

An alternative is constructed, `forkAlt` {16, 17}. Once a fork has been picked up by a philosopher it can only be put down by that philosopher, thus all we have to do is process the signal indicating the picking up of the fork {20} and then wait for the signal indicating that it has been put down {21}.

The college has employed a lazy butler who simply notes the entries and exits to the dining room and does little else apart from washing the forks and replenishing the bowl of spaghetti. The latter actions are of no concern. The behaviour of the `LazyButler` is shown in Listing 12-3.

```
10 class LazyButler implements CSProcess {
11
12   def ChannelInputList enters
13   def ChannelInputList exits
14
15   void run() {
16     def seats = enters.size()
17     def allChans = []
18
19     for ( i in 0 ..< seats ) { allChans << exits[i] }
20     for ( i in 0 ..< seats ) { allChans << enters[i] }
21
22     def eitherAlt = new ALT ( allChans )
23
24     while (true) {
25       def i = eitherAlt.select()
26       allChans[i].read()
27     } // end while
28   } //end run
29 } // end class
```

Listing 12-3 The Lazy Butler's Behaviour

The channels used to signal the entry and exit from the room are passed to the `LazyButler` as `ChannelInputLists` `enters` {12} and `exits` {13}. The number of `seats` in the dining room can be determined by the size of the list `enters` {16}. A List of all the channels, `allChans` {17} is defined to which each of the elements of the `exits` and `enters` lists are appended {19, 20}. An alternative, `eitherAlt` is defined over `allChans` {22} and as signals are received {25} on any of the channels they are read {26} and ignored by the lazy butler.

The college, believing this to be a sufficient solution, implements it as shown in Listing 12-4 in the script `c12.fork.RunLazyCollege`.

```
10 def PHILOSOPHERS = 5
11
12 def lefts = Channel.one2oneArray(PHILOSOPHERS)
13 def rights = Channel.one2oneArray(PHILOSOPHERS)
14 def enters = Channel.one2oneArray(PHILOSOPHERS)
15 def exits = Channel.one2oneArray(PHILOSOPHERS)
16
17 def entersList = new ChannelInputList(enters)
18 def exitsList = new ChannelInputList(exits)
19
20 def butler = new LazyButler ( enters: entersList, exits: exitsList )
21
```

```
22 def philosophers = ( 0 ..< PHILOSOPHERS).collect { i ->
23           return new Philosopher ( leftFork: lefts[i].out(),
24                                    rightFork: rights[i].out(),
25                                    enter: enters[i].out(),
26                                    exit: exits[i].out(), id:i ) }
27
28 def forks = ( 0 ..< PHILOSOPHERS).collect { i ->
29  return new Fork ( left: lefts[i].in(),
30  right: rights[(i+1)%PHILOSOPHERS].in() ) }
31
32 def processList = philosophers + forks + butler
33
34 new PAR ( processList ).run()
```

Listing 12-4 The College's Lazy Implementation

The number of `PHILOSOPHERS` is defined {10} and then each of the required channel arrays {12–15} and corresponding channel lists {17, 18} are defined. The `butler` and collection of `philosophers` are defined, passing channel parameters as required {20–26}. The collection of `forks` is then defined {28–30} noting that the same fork can be accessed as the left fork of the i'th philosopher and the right fork of the i+1'th philosopher {30}, using modulo arithmetic to ensure the subscripts stay in range. Execution of this scheme produces the output shown in Output 12-1.

```
            thinking : 1
            thinking : 2
            thinking : 3
            thinking : 4
            thinking : 0
1: entered
2: entered
3: entered
4: entered
0: entered
2: got left fork
3: got left fork
1: got left fork
4: got left fork
0: got left fork
```

Output 12-1Operation Of The Lazy College

As can be observed, all the philosophers think, then enter the dining room and then they each pick up their left fork after which no further progress is possible. Even more worrying for the college is that sometimes the solution appears to work. Faced with this situation the college reflects on their operation and decides that the butler has to be more proactive in managing the dining room.

12.2 Proactive Management

The butler is now required to ensure that no more than four of the philosophers are in the room at any one time. This guarantees that at least one of the philosophers will be able to pick the single spare fork on their right hand side. The required behaviour of the butler is shown in Listing 12-5.

The first part of the behaviour up to {21} is identical to that of the `LazyButler` except that a variable `seated` has been defined {17}, which counts the number of philosophers already sitting. In addition, an extra alternative, `exitAlt` {24} is defined over the exits only. Initially, the butler determines whether there are at least two spare seats in the room {27}, in which case there is `space` for another philosopher to enter and start eating. In this case we can accept an input on any of the channels, `allChans`, managed by the butler. If there is no space then we can only accept inputs from philosophers wishing to exit the room. The alternative to use is determined based on the value of `space` {28}. An enabled input is then `selected` {29} and `read` {30}. It is important to note that `allChans` contains the `exits` channels first so that we can read exit signals from `allChans`; regardless of which alternative is used. We can determine whether or not this instance results from a philosopher exiting or entering the room by testing the index of the read channel, `i`, against the number of `seats` {31} and updating the number of philosophers seated accordingly.

The college is very relieved to discover that this simple change of butler behaviour is sufficient to remedy the situation provided they replace the invocation of the `LazyButler` by the `Butler` on line {20} of Listing 12-4.

```
10 class Butler implements CSProcess {
11
12   def ChannelInputList enters
13   def ChannelInputList exits
14
15   void run() {
16     def seats = enters.size()
17     def seated = 0
18
19     def allChans = []
20     for ( i in 0 ..< seats ) { allChans << exits[i] }
21     for ( i in 0 ..< seats ) { allChans << enters[i] }
22
23     def eitherAlt = new ALT ( allChans )
24     def exitAlt = new ALT ( exits )
```

```
25
26      while (true) {
27        def spaces = seated < ( seats - 1 )
28        def usedAlt = spaces ? eitherAlt : exitAlt
29        def i = usedAlt.select()
30        allChans[i].read()
31        def exiting = i < seats
32        seated = exiting ? seated - 1 : seated + 1
33      } // end while
34    } //end run
35 } // end class
```

Listing 12-5 The Modified Butler Behaviour

Output from the modified butler behaviour is shown in Output 12-2. It can be seen that all bar Philosopher 0 enter the room and that means that Philosopher 1 and 2 can eat at the same time. When Philosopher 1 finishes eating and leaves the room to resume thinking, Philosopher 3 is now able to eat. Further analysis shows that there are two Philosophers eating most of the time as should be expected. Thinking appears to be a solitary activity!

```
             thinking : 0
             thinking : 1
             thinking : 2
             thinking : 3
             thinking : 4
1: entered
2: entered
3: entered
4: entered
1: got left fork
2: got left fork
3: got left fork
4: got left fork
1: got right fork
             eating : 1
2: got right fork
             eating : 2
1: put down left
1: put down right
0: entered
0: got left fork
1: exited
             thinking : 1
2: put down left
3: got right fork
             eating : 3
```

Output 12-2 Modified Behaviour

12.3 A More Sophisticated Canteen

In an effort to provide a better service the college decides that, rather than having a single dining room with its somewhat limited eating facilities, it is going to invest in a canteen style food facility. Philosophers will be allowed to enter the canteen, go to a serving hatch, pick up their food, in the form of a chicken, without having to wait, in fact waiting will not be allowed and then go into the canteen to find a place to sit. The college authorities guarantee that there will be sufficient places for everyone to sit and that nothing else can go wrong. They are so confident that they allow any number of philosophers to enter the canteen. To this end they have decided that a visual display will be provided showing the state of the kitchen, in which the chef cooks the chickens, the state at the serving hatch and they have also installed monitoring devices that shows the action each philosopher is currently undertaking.

The chef is capable of cooking four chickens at a time but it does take time for them to cook and also to take them to the serving hatch. This is shown in Listing 12-6.

```
10 class Chef implements CSProcess {
11
12   def ChannelOutput supply
13   def ChannelOutput toConsole
14
15   void run () {
16
17     def tim = new CSTimer()
18     def CHICKENS = 4
19
20     toConsole.write( "Starting … \n")
21     while(true){
22       toConsole.write( "Cooking … \n") // cook 4 chickens
23       tim.after (tim.read () + 2000) // this takes 2 seconds to cook
24       toConsole.write( "$CHICKENS chickens ready … \n")
25       supply.write (CHICKENS)
26       toConsole.write( "Taking chickens to Canteen … \n")
27       supply.write (0)
28     }
29   }
30 }
```

Listing 12-6 The Chef's behaviour

The `supply` channel {12} is used to indicate to the canteen how many chickens are about to arrive. The `toConsole` channel {13} is used to write information on the display. It takes 2 seconds to cook the chickens {23} with appropriate messages output to the console. The number of chickens is sent on the `supply` channel to the canteen {25}. The write to the supply channel {26} is used to represent the point at which the chickens have been transferred to the serving hatch as can be seen in Listing 12-7.

The canteen receives requests for a chicken from a philosopher on the `service` channel {12} and notification of its availability is given on the `deliver` channel {13}. The `Chef` process uses the `supply` channel to indicate that chickens are ready for serving {14}. The `toConsole` channel is used to display the current availability of chickens on the display {15}. The canteen alternates over the `supply` and `service` channels {19}. A `timer` {24} is required to reflect the time it takes to set down the chickens by the `Chef`. The enabled alternative is `select`ed using the `fair` option {30}.

In the case of `SUPPLY`, when more chickens become available, the `value` is read from `supply` {32} and a message written to the console {33}. A delay of 3 seconds is created {34} representing the time taken to transfer chickens from the kitchen to the canteen. After this the number of chickens available is incremented {35} by `value`. The canteen console is updated {36} and the signal written by the `Chef` {Listing 12-6, 27} is read {37} and this permits the `Chef` to return to the `Kitchen` to cook more chickens.

```
10 class InstantCanteen implements CSProcess {
11
12   def ChannelInput service
13   def ChannelOutput deliver
14   def ChannelInput supply
15   def ChannelOutput toConsole
16
17   void run () {
18
19     def canteenAlt = new ALT ([supply, service])
20
21     def SUPPLY = 0
22     def SERVICE = 1
23
24     def tim = new CSTimer()
25     def chickens = 0
26
27     toConsole.write( "Canteen : starting … \n")
28
29     while (true) {
30       switch (canteenAlt.fairSelect ()) {
31         case SUPPLY:
32           def value = supply.read()
33           toConsole.write( "Chickens on the way …\n")
```

```
34            tim.after (tim.read() + 3000)
35            chickens = chickens + value
36            toConsole.write( "$chickens chickens now available …\n")
37            supply.read()
38          break
39          case SERVICE:
40            def id = service.read()
41            if ( chickens > 0 ) {
42              chickens = chickens - 1
43              toConsole.write ("chicken ready for Philosopher $id …$chickens
                                  chickens left\n")
44              deliver.write(1)
45            }
46            else {
47              toConsole.write( " NO chickens left … \n")
48              deliver.write(0)
49            }
50          break
51        }
52      }
53    }
54 }
```

Listing 12-7 The Canteen Behaviour

When a philosopher requires `SERVICE`, their `id` is `read` from the `service` channel {40}. The `Canteen` at this point recognises that there may be no chickens available but is sure that this will not happen. Thus a test is undertaken on the number of available `chickens` {41} and if there is a chicken available the number of `chickens` is decremented {42} and a message to that effect output {43}. The philosopher is informed by the writing of a 1 on the `deliver` channel {44}. If no chickens are available, a message is displayed {47} and a zero is written to the deliver channel {48}.

The behaviour of the Philosophers is now somewhat different; they still think and eat forever, in rotation. However, the philosophers are now somewhat sanguine about the College authorities' capabilities and use a behaviour in which they try to cover every eventuality as shown in Listing 12-8. A philosopher has an `id` {12}, a channel upon which a `service` request is made {13} and one upon which a chicken `delivery` is made {14} plus a channel to write messages on a console {15}. A `timer` {18} is required to time the philosopher's actions and an initial message is written `toConsole` {19}.

```
10 class PhilosopherBehaviour implements CSProcess {
11
12   def int id = -1
13   def ChannelOutput service
14   def ChannelInput deliver
15   def ChannelOutput toConsole
16
```

```
17    void run() {
18      def tim = new CSTimer()
19      toConsole.write( "Starting … \n")
20      while (true) {
21        toConsole.write( "Thinking … \n")
22        if (id > 0) {
23          tim.sleep (3000)
24        }
25        else {
26          // Philosopher 0, has a 0.1 second think
27          tim.sleep (100)
28        }
29        toConsole.write( "Need a chicken …\n")
30        service.write(id)
31        def gotOne = deliver.read()
32        if ( gotOne > 0 ) {
33          toConsole.write( "Eating … \n")
34          tim.sleep (2000)
35          toConsole.write( "Brrrp … \n")
36        }
37        else {
38          toConsole.write( "              Oh dear No chickens left \n")
39        }
40      }
41    }
42 }
```

Listing 12-8 The Philosopher Behaviour

Initially, a philosopher thinks for 3 seconds {22}, unless they are philosopher 0 who only thinks for 0.1 seconds {27}. At this point the behaviour is common and starts by indicating on the console that the philosopher needs a chicken {29}, and is followed by a signal request on the service channel with the philosopher's id {30}. At this point we note that the philosopher is behaving like a client and thus immediately follows the service request with the input of the chicken on the `deliver` channel {31} containing the server response from the canteen. The philosopher now tests the value of `gotOne` {32} to see if they have been given a chicken. If this is the case, then a message is output and the philosopher takes 2 seconds to eat the chicken, after which he burps {35}. If no chicken is available a sad message appears {38}.

The above process is formed into a further process each with a `GConsole`, upon which console messages can be displayed.

The script that invokes the system is shown in Listing 12-9. The channels that implement the `service` and `deliver` connections between the philosophers and the canteen are shared {10, 11}, `any2one` and `one2any` channels respectively, enabling any of the philosophers to access the canteen. A list of five philosophers is then created with each connected to `service` and `deliver` {16, 17}. The other processes, `InstantServery` comprising the canteen and its console and the `Kitchen` comprising the `Chef` and its console are added to `processList` {20-24}. The processes are then run. This can be observed by running the script `InstantCollege` in `c12.examples.canteen`. Needless to say we observe that some philosophers do not get a chicken and more importantly miss their turn!

```
10 def service = Channel.any2one ()
11 def deliver = Channel.one2any ()
12 def supply = Channel.one2one ()
13
14 def philosopherList = (0 .. 4).collect{
15           i -> return new Philosopher( philosopherId: i,
16                                        service: service.out(),
17                                        deliver: deliver.in())
18           }
19
20 def processList = [ new InstantServery ( service:service.in(),
21                                        deliver:deliver.out(),
22                                        supply:supply.in()),
23                     new Kitchen (supply:supply.out())
24                  ]
25
26 processList = processList + philosopherList
27 new PAR ( processList ).run()
```

Listing 12-9 The Instant Canteen Script

It is obvious that the behaviour of the canteen is at fault as it did not stop philosophers making requests for service when there were no chickens available. The revised behaviour is shown in Listing 12-10, which has been augmented by the use of pre-conditions.

The precondition array is initialised {20} so that chickens can always be supplied from the kitchen. Initially, there are no chickens available so the `service` precondition is `false`. At the start of the process' main loop the state of the service precondition is re-evaluated {31}. If no chickens are available a message to that effect is displayed {32–34}. Now, of course, we enter each case in the `switch` associated with the enabled alternative knowing the precise state of the canteen and thus the coding is much simpler. In particular, we only permit `service` requests when we are assured that chickens are available {44–48}.

This version of the system can be executed using the script `QueuingCollege` and another version that shows clock ticks in the canteen console is also available, `ClockedQueuingCollege`. It can be observed from an execution of the system, which allows numbers other than five philosophers, that every philosopher gets a chicken whenever they are hungry, however, they may have to wait.

```
10 class QueuingCanteen implements CSProcess {
11
12   def ChannelInput service
13   def ChannelOutput deliver
14   def ChannelInput supply
15   def ChannelOutput toConsole
16
17   void run () {
18
19     def canteenAlt = new ALT ([supply, service])
20     def boolean [] precondition = [true, false ]
21
22     def SUPPLY = 0
23     def SERVICE = 1
24
25     def tim = new CSTimer()
26     def chickens = 0
27
```

```
28      toConsole.write ("Canteen : starting … \n")
29
30      while (true) {
31        precondition[SERVICE] = (chickens > 0)
32        if (chickens == 0 ){
33          toConsole.write ("Waiting for chickens …\n")
34        }
35        switch (canteenAlt.fairSelect (precondition)) {
36          case SUPPLY:
37            def value = supply.read()
38            toConsole.write ("Chickens on the way …\n")
39            tim.after (tim.read() + 3000)
40            chickens = chickens + value
41            toConsole.write ("$chickens chickens now available …\n")
42            supply.read()
43          break
44          case SERVICE:
45            def id = service.read()
46            chickens = chickens - 1
47            toConsole.write ("chicken ready for Philosoper $id … $chickens
                                chickens left \n")
48            deliver.write(1)
49          break
50        }
51      }
52    }
53 }
```

Listing 12-10 The Revised Canteen With Alternative Pre-conditions

12.4 Summary

This chapter has presented solutions to the classical dining philosophers' problem using two different formulations. The second solution, using a canteen is also an instance of the client-server design pattern with the canteen acting as a pure server and the chef and philosophers acting as pure clients. This perhaps demonstrates that even though the coding in both cases followed the client-server pattern it was still possible to create an erroneous solution. The client-server design pattern is not a panacea for all occasions; it has to be applied sensibly and with understanding. Even if the communication patterns are correct it is still possible to create incorrect systems if insufficient thought is given to the problem solution.

13 Accessing Shared Resources: CREW

Shared resource management is commonly used to access large amounts of data by many users;

- the shared resource concept is defined
- concurrent read exclusive write is explained
- a simple example is developed demonstrating the concept

This chapter describes techniques that were developed for, and are used most often in shared memory multi-processing systems. In such systems great care has to be taken to ensure that processes running on the same processor do not access an area of shared memory in an uncontrolled manner. Up to now the solutions have simply ignored this problem because all data has been local to and encapsulated within a process. One process has communicated data to another as required by the needs of the solution. The process and channel mechanisms have implicitly provided two capabilities, namely synchronisation between processes and mutual exclusion of data areas. In shared memory environments the programmer has to be fully aware of both these aspects to ensure that neither is violated.

Mutual exclusion ensures that while one process is accessing a piece of shared data no other process will be allowed access regardless of the interleaving of the processes on the processor. Synchronisation ensures that processes gain access to such shared data areas in a manner that enables them to undertake useful work. The simplest solution to both these problems is to use a pattern named CREW, Concurrent Read Exclusive Write, which, as its names suggests, allows any number of reader processes to access a piece of shared data at the same time but only one writer process to access the same piece of data at one time. The CREW mechanism manages this requirement and in sensible implementations also imposes some concept of fairness. If access is by multiple instances of reader and writer processes then one could envisage a situation where the readers could exclude writers and vice versa and this should be ameliorated as far as is possible. The JCSP implementation of a CREW does exhibit this capability of fairness, as shall be demonstrated.

At the simplest level the CREW has to be able to protect accesses to the shared data and the easiest way of doing this is to surround each access, be it a read or write with a call to a method that allows the start of an operation and subsequently when the operation is finished to indicate that it has ended. Between such pairs of method calls the operation of the CREW is guaranteed. Thus the programmer has to surround access to shared data with the required start and end method calls be they a read or write to the shared data. It is up to the programmer to ensure that all such accesses to the shared data are suitably protected.

In the JCSP implementation of CREW we extend an existing storage collection with a `Crew` class. Then we ensure that each access that `put`s data into the collection is surrounded by a `startWrite`() and `endWrite`() pair of method calls on the `Crew`. Similarly, that each `get` access is surrounded by a `startRead`() and `endRead`() method call. Internally, the `Crew` then ensures that access to the shared storage collection is undertaken in accordance with the required behaviour. Further, fairness can be implemented quite simply by ensuring that if the shared data is currently being accessed by one or more reader processes then as soon as a writer process indicates that it wishes to put some data into the shared collection then no further reader processes are permitted to start reading until the write has finished. Similarly, a sequence of write processes, each of which requires exclusive access, will be interposed by reader process accesses as necessary.

13.1 CrewMap

Listing 13-1 shows a simple extension of a `HashMap` {10} by means of an instance of a `Crew` {12}. The `put` and `get` methods of `HashMap` are then overwritten with new versions that surround them with the appropriate start and end method calls {15, 17} and {21, 24}, between which the normal `HashMap`'s `get` and `put` methods can be called as usual.

```
10 class CrewMap extends HashMap<Object, Object> {
11
12   def theCrew = new Crew()
13
14   def Object put ( Object itsKey, Object itsValue ) {
15     theCrew.startWrite()
16     super.put ( itsKey, itsValue )
17     theCrew.endWrite()
18   }
19
20   def Object get ( Object itsKey ) {
21     theCrew.startRead()
22     def result = super.get ( itsKey )
23     theCrew.endRead()
24     return result
25   }
26
27 }
```

Listing 13-1 The CrewMap Class Definition

At this point a word of caution has to be given. This arises because Java allows exceptions to be thrown at any point. Thus in the above formulation it might be possible for the lines that represent normal access to the shared resource {16, 22} to fail. In such a case the call to the `end` synchronisation method {17, 23} will never happen and thus the `Crew` will fail in due course as the required locks will not be released. The associated documentation for JCSP Crew discusses this in more detail. The solution is to encapsulate the access in a `try .. catch .. finally` block. The problem arises because Java invokes code sequences that are not part of the coding sequence and thus the programmer has to be very wary of these possibilities. In the following description we shall presume that all access is well behaved and such a fault will not occur.

Once the `CrewMap` has been defined it can be used in a solution that requires multiple processes access to its shared data collection. Figure 13-1 shows such a typical application. In this case two `Read` and two `Write` processes access the shared `DataBase` resource. The coding of the `DataBase` process is shown in Listing 13-2.

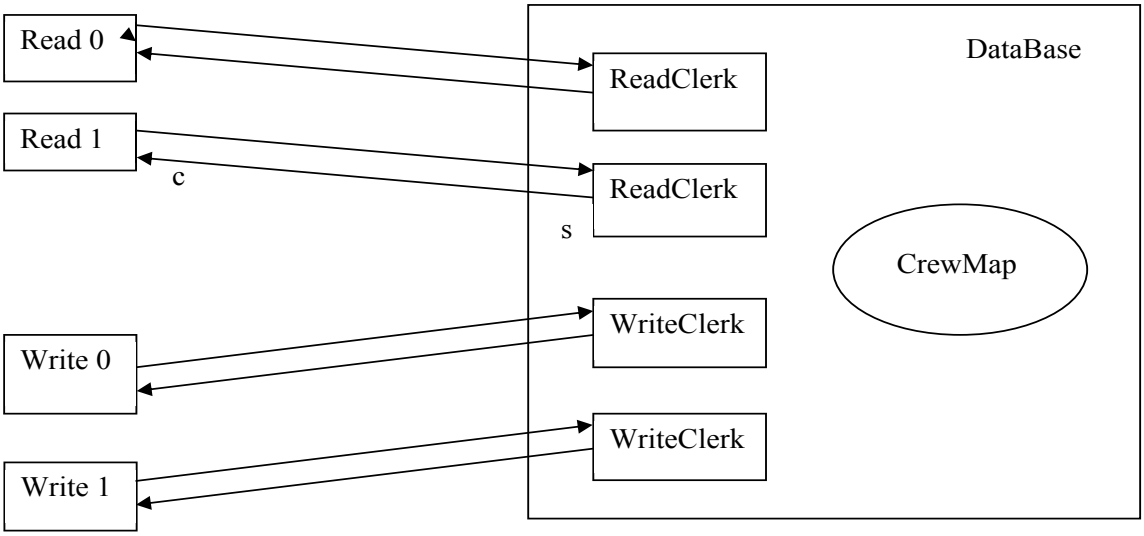


Figure 13-1 A Simple Use of CrewMap

13.2 The DataBase Process

The `DataBase` process has two channel list properties {12, 13} comprising the channels used by the `Read` and `Write` processes to access it. Additionally, properties are required that define the number of such `Read` and `Write` processes, `readers` and `writers` respectively {14, 15}.

```
10 class DataBase implements CSProcess {
11
12   def ChannelInputList inChannels
13   def ChannelOutputList outChannels
14   def int readers
15   def int writers
16
17   void run () {
18     println "DataBase has started"
19     def crewDataBase = new CrewMap()
20     for ( i in 0 ..< 10 ) {
21       crewDataBase.put ( i, 100 + i)
22     }
23     for ( i in 0 ..< 10 ) {
24         println "DB: Location $i contains ${crewDataBase.get( i )} "
25     }
26     def processList = []
27     for (i in 0..< readers) {
28         processList.putAt (i, new ReadClerk ( cin: inChannels[i],
29                                               cout: outChannels[i],
30                                               data: crewDataBase) )
31     }
32     for ( i in 0 ..< writers ) {
33         processList.putAt ( ( i + readers), new WriteClerk (cin:
           inChannels[i + readers],
34                                  cout: outChannels[i + readers],
35                                  data: crewDataBase ) )
36     }
37     new PAR (processList).run()
38   }
39 }
```

Listing 13-2 The DataBase Process definition

The `run` method {17} essentially creates the structure shown in Figure 13-1. An instance of `CrewMap` is defined called `crewDataBase` {19}. The shared resource `crewDataBase` is then populated with initial values {20–22}, which initialises the first ten locations with the values 100 to 109 in sequence. An empty `processList` {26} is then defined that will hold instances of the required `ReadClerk` and `WriteClerk` processes. The required number of `ReadClerk` processes are then created {27–36} and placed in `processList`. Each `ReadClerk` is allocated the corresponding element of the `inChannels` and `outChannels` channel lists {28, 29}. Finally, the `ReadClerk` process has its `data` property initialised to the `crewDataBase` itself {30}. The `WriteClerk` processes are instantiated in the same manner {32–36} ensuring that the correct elements of the `inChannels` and `outChannels` lists are allocated to the processes. This means that the all the `ReadClerk` and `WriteClerk` processes have shared access to the `crewDataBase`. The `processList` can now be passed to a `PAR` for `run`ning {37}.

Communication between the `Read` and `Write` processes and the `DataBase` is achieved by a single class called `DataObject` {10}, see Listing 13-3. `DataObject` comprises three properties {12–14}, `pid` hold the identity number of the accessing `Read` or `Write` process, `location` holds the index of the resource element to be accessed and `value` is either the value read from that element or that is to be written to the element.

```
10 class DataObject implements Serializable, JCSPCopy {
11
12   def int pid
13   def int location
14   def int value
15 }
```

Listing 13-3 The Definition of DataObject (Omitting Methods copy and toString)

It should be noted that this formulation of the `DataBase` contains no alternative (`ALT`) as might be expected from previous examples. This arises because we are using a formulation that contains a CREW that essentially provides the same functionality, but only for shared memory applications. The advantage of the alternative is that it can be used to alternate over networked channels and thus is more flexible. It also has the advantage of exposing the alternative concept that is so important in the modelling of parallel systems.

13.3 The Read Clerk Process

Listing 13-4 shows the `ReadClerk` process, which has channel input and output properties `cin` {12} and `cout` {13} respectively and a `data` property {14} that accesses the CREW resource.

```
10 class ReadClerk implements CSProcess {
11
12   def ChannelInput cin
13   def ChannelOutput cout
14   def CrewMap data
15
16   void run () {
17     println "ReadClerk has started "
18     while (true) {
19       def d = new DataObject()
20       d = cin.read()
21       d.value = data.get ( d.location )
22       println "RC: Reader ${d.pid} has read ${d.value} from ${d.location}"
23       cout.write(d)
24     }
25   }
26 }
```

Listing 13-4 The ReadClerk Process

The `run` method {16-25} defines an instance `d` of type `DataObject` {19} after which the value of `d` is read from `cin` {20}. The `location` property of `d` is then used to access the `CrewMap` property `data` {21} to `get` the corresponding value which is then stored in the `value` property of `d`. The revised value of `d` is then written to the channel `cout` {23}, after an appropriate message is printed.

13.4 The Write Clerk Process

The `WriteClerk` process is shown in Listing 13-5 and is fundamentally the same as that shown in the `ReadClerk` process except that a new `value` is `put` into the shared resource {21}. The unmodified `DataObject d` is written back to the corresponding `Write` process to confirm that the operation has taken place {23}.

```
10 class WriteClerk implements CSProcess {
11
12   def ChannelInput cin
13   def ChannelOutput cout
14   def CrewMap data
15
16   void run () {
17     println "WriteClerk has started "
18     while (true) {
19       def d = new DataObject()
20       d = cin.read()
```

```
21        data.put ( d.location, d.value )
22        println "WC: Writer ${d.pid} has written ${d.value} to ${d.location}"
23        cout.write(d)
24      }
25    }
26 }
```

Listing 13-5 The WriteClerk Process

Each of the Clerk processes behaves as a pure server. The server behaviour is guaranteed provided access to the shared data resource always complete in finite time. This will happen provided no exception is thrown and handled incorrectly in the shared data resource.

13.5 The Read Process

The `Read` process is shown in Listing 13-6. It has three properties. A channel by which it writes to the database `r2db` {12} and one by which it reads returned values `db2r` {13}. The last property, `id` {14}, is the identity number of the `Read` process. The channel `toConsole` {15} writes messages to an associated `GConsole` process. The `run` method {17} initialises a `DataObject` with the `Read` process' `id` {22} and then reads a value from each location of the shared resource in sequence {20}, printing out each returned value {25}. This is achieved by allocating the loop value `i` to the `location` property of `d` {22}. The instance `d` is then written to the shared resource using the channel `r2db` {23}. The process then waits until it can read the returned `DataObject` into `d` using the channel `db2r` {24}. This means that the process behaves as a pure client.

```
10 class Read implements CSProcess {
11
12   def ChannelOutput r2db
13   def ChannelInput db2r
14   def int id
15   def ChannelOutput toConsole
16
17   void run () {
18     def timer = new CSTimer()
19     toConsole.write ( "Reader $id has started \n")
20     for ( i in 0 ..<10 ) {
21       def d = new DataObject(pid:id)
22       d.location = i
23       r2db.write(d)
24       d = db2r.read()
25       toConsole.write ( "Location "d.location+" has value "+d.value + "\n")
26       timer.sleep(100)
27     }
28     toConsole.write ( "Reader $id has finished \n")
29   }
30 }
```

Listing 13-6 The Read Process

13.6 The Write Process

The `Write` process is shown in Listing 13-7 and is very similar to the `Read` process except that the elements of the shared resource are accessed in reverse order, that is from 9 to 0 {22}. The value written to the shared resource is dependent upon the `id` of the writing process {24} and is sufficiently different to make observation of the resulting behaviour easier.

```
10 class Write implements CSProcess {
11
12   def ChannelOutput w2db
13   def ChannelInput db2w
14   def int id
15   def ChannelOutput toConsole
16
17   void run () {
18     def timer = new CSTimer()
19     toConsole.write ( "Writer $id has started \n" )
20     for ( j in 0 ..<10 ) {
21       def d = new DataObject(pid:id)
22       def i = 9 - j // write in reverse order
23       d.location = i
24       d.value = i + ((id+1)*1000)
25       w2db.write(d)
26       d = db2w.read()
27       toConsole.write ("Location "+d.location+" now contains "+d.value+"\n")
28       timer.sleep(100)
29     }
30     toConsole.write ( "Writer $id has finished \n" )
31   }
32 }
```

Listing 13-7 The Write Process

13.7 Creating the System

The script that invokes the `DataBase` system is shown in Listing 13-8.

```
10 def nReaders = Ask.Int ( "Number of Readers ? ", 1, 5)
11 def nWriters = Ask.Int ( "Number of Writers ? ", 1, 5)
12 def connections = nReaders + nWriters
13
14 def toDatabase = Channel.one2oneArray(connections)
15 def fromDatabase = Channel.one2oneArray(connections)
16 def consoleData = Channel.one2oneArray(connections)
17
18 def toDB = new ChannelInputList(toDatabase)
19 def fromDB = new ChannelOutputList(fromDatabase)
20
```

```
21 def readers = ( 0 ..< nReaders).collect { r ->
22                       return new Read (id: r,
23                                        r2db: toDatabase[r].out(),
24                                        db2r: fromDatabase[r].in(),
25                                        toConsole: consoleData[r].out())
26                       }
27
28 def writers = ( 0 ..<nWriters).collect { w ->
29                       int wNo = w + nReaders
30                       return new Write ( id: w,
31                                        w2db: toDatabase[wNo].out(),
32                                        db2w: fromDatabase[wNo].in(),
33                                        toConsole: consoleData[wNo].
                                          out())
34                       }
35
36 def database = new DataBase ( inChannels: toDB,
37                              outChannels: fromDB,
38                              readers: nReaders,
39                              writers: nWriters)
40
41 def consoles = ( 0 ..< connections).collect { c ->
42                       def frameString = c < nReaders ?
43                                 "Reader " + c :
44                                 "Writer " + (c - nReaders)
45                       return new GConsole (toConsole: consoleData[c].in(),
46                                        frameLabel: frameString )
47                       }
48 def procList = readers + writers + database + consoles
49
50 new PAR(procList).run()
```

Listing 13-8 The Script to Invoke the DataBase System

Initially, the number of `Read` and `Write` processes is obtained {10, 11} by a console interaction. The total number of connections to the `DataBase` is then calculated as `connections` {12}. The system uses a `GConsole` process for each `Read` and `Write` process to display the outcome of the interactions with the `DataBase`. The channels used to connect the `Read` and `Write` processes to the `Database` and the `GConsoles` are then defined {14–16}. The corresponding channel lists `toDb` and `fromDb` are then defined {18, 19}, which connect the `Read` and `Write` processes to the `DataBase`.

The required number of `Read` processes is then created in the list `readers` {21–26}. Each instance uses the closure property `r` to identify the required element of the previously declared channel arrays that connect the process to the `DataBase` and its `GConsole` process. Similarly, the required number of Write processes is defined {28–34}. The variable `wNo` {29} is used to ensure that the index used to associate `Write` process channel indeces is offset by the number of `Read` processes.

An instance of the `DataBase` process is then created {36–39}, using the previously declared channel lists. The list `consoles` {41–47} contains the instances of `GConsole` required to connect to the `Read` and `Write` processes. Finally, `procList` is created as the addition of all the process lists and the `database` process {48} and then `run` {50}.

Outputs 13-1 and 13-2 show the output from the running of the system when it is started with two `Read` and two `Write` processes. The order in which the `Write` process have been executed can be determined from the values that have been read by the two `Read` processes. Recall that the `Write` processes access the database locations in reverse order to the `Read` processes. The outputs indicate that the implementation of the Crew class is inherently fair because the values read by the `Read` processes change from the initial values to the modified values about half way through the cycle. The values read from locations 5 and 6 also vary indicating that state of the `DataBase` was in flux at that point in the access cycles with read and write operations fully interleaved.

```
Location 1 has value 101
Location 2 has value 102
Location 3 has value 103
Location 4 has value 104
Location 5 has value 105
Location 6 has value 1006
Location 7 has value 2007
Location 8 has value 2008
Location 9 has value 2009
Reader has finished
```

Output 13 – 1 Output From Read process 0

```
Location 1 has value 101
Location 2 has value 102
Location 3 has value 103
Location 4 has value 104
Location 5 has value 1005
Location 6 has value 2006
Location 7 has value 2007
Location 8 has value 2008
Location 9 has value 2009
Reader has finished
```

Output 13 – 2 Output From Read process 1

13.8 Summary

In this chapter we have investigated a typical mechanism used in shared memory multi-processing system. The formulation tends to hide the interactions that take place because these are captured somewhat remotely in the `CrewMap` class definition.

13.9 Challenge

Rewrite the system so that a Crew is not used and the DataBase process alternates over the input channels from the Read and Write processes. The system should capture the same concept of fairness as exhibited in the CREW based solution.

14 Barriers and Buckets: Hand-Eye Co-ordination Test

This chapter develops a solution to a highly dynamic system using a number of shared meory synchronisation capabilities including:

- barrier
- alting barrier
- bucket
- channel data stores are used to overcome inconsistenices in the underlying Java user interface model

Three shared memory synchronisation techniques are combined to provide control of a highly dynamic environment. A `Barrier` provides a means whereby a known number of processes collectively control their operation so they all wait at the barrier until all of them have synchronised with the barrier at which time they are all released to run in parallel. An `AltingBarrier` is a specialisation of the `Barrier` that allows it to act also as a guard in an Alternative (Welch, et al., 2010). Finally, a `Bucket` (Kerridge, et al., 1999) provides a flexible refinement of a barrier. Typically, there will be a collection of `Bucket`s into which processes are placed depending upon some criterion. Another process then, subsequently, causes a `Bucket` to flush all its processes so they are executed concurrently. These processes will in due course, become idle, whereupon they place themselves in other buckets. The next `Bucket` in sequence is then flushed and so the cycle is repeated. `Bucket`s can be used to control discrete event simulations in a very simple manner. The process that undertakes the flushing of the buckets must not be one of the processes that can reside in a `Bucket`.

The aim of this example is to present a user with a randomly chosen set of targets that each appear for a different random time. During the time the targets are available the user clicks the mouse over each of the targets in an attempt to hit as many of the targets as possible. The display includes information of how many targets have been hit and the total number of targets that have been displayed. The targets are represented by different coloured squares on a black background and a hit target is coloured white. A target that is not 'hit' before its self determined random time has elapsed is coloured grey. There is a gap between the end of one set of targets and the display of the next set during which time the screen is made all black. The minimum time for which a target is displayed is set by the user; obviously the longer this time the easier it is to hit the targets. Targets will be available for a period between the shortest time and twice that time. Figure 14-1 shows the screen, at the point when six targets have been displayed, and none have yet been hit. The system has displayed a total of 88 targets of which 15 targets have been hit. The minimum target delay was 900 milliseconds. It can be deduced there are 16 targets in a 4×4 matrix.

The solution presumes that each target is managed by its own process and that it is these processes that are held in a `Bucket` until it is the turn of that `Bucket` to be flushed. When a target is enabled it displays itself until either it is 'hit' by a mouse-press, in which case it turns white, or the time for which it appears elapses and it is coloured grey. It is obvious that each of these target processes will finish at a different time and because the number of targets is not predetermined a barrier is used to establish when all the enabled target processes have finished. After this, the target process determines into which bucket it is going fall and thereby remains inactive until that bucket is flushed. The other processes used in the solution are shown in Figure 14-2.

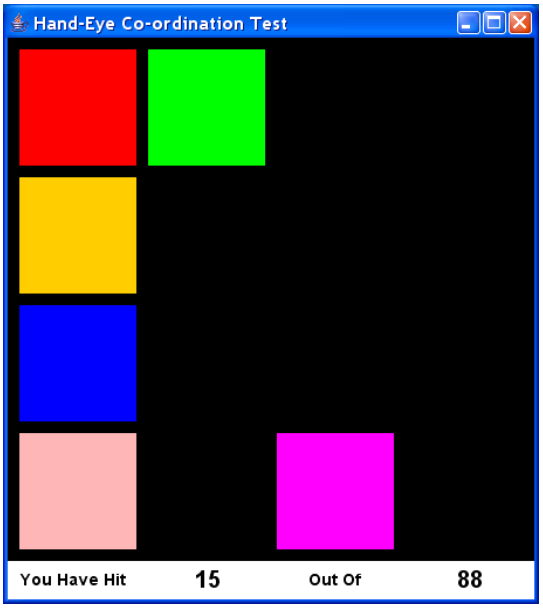


Figure 14-1 The Screen for the Hand-Eye Co-ordination Test

The system comprises a number of distinct phases each of which is controlled by its own barrier, which depending on the context is either a simple `Barrier` or an `AltingBarrier`.

Figure 14-2 shows the system at the point where it is about to synchronise on the `setUpBarrier`. During this setup phase there are no channel communications but the processes that synchronise on `setUpBarrier` either have to initialise themselves in some manner or must not progress beyond a certain point to ensure the system will not get out of step with itself. The setup phase only occurs once when the system is initially executed. The processes that are not part of the `setUpBarrier` cannot make any progress because they are dependent on other barriers or communications with processes that synchronise on the `setUpBarrier`.

The `BarrierManager` is a process that is used to manage phase synchronisations and as such will be seen in subsequent figures to be part of a number of other barriers. For ease of description the structure of each phase will show only the relevant barrier and channels that are operative at that time. The separation into these distinct phases also makes it easier to analyse the system from the point of view of its client-server architecture, thereby enabling deadlock and livelock analysis.

The `TargetFlusher` and `TargetProcess` processes are the only processes that can manipulate the array of `Buckets`. The `Buckets` are not shown on the diagram. The `TargetProcesses` are able to identify which `Bucket` they are going to enter when they stop running. `TargetFlusher` is the only process that can cause the flush and subsequent execution of the processes contained with a `Bucket`. The processing cycle of a `TargetProcess` is to wait until it is flushed from a `Bucket`; it then runs until it determines, itself, that it has ceased to run at which point it causes itself to `fallInto` a `Bucket`, which it also determines.

The `DisplayController` process initialises the display window to black. It also initialises, to zero, the information contained in the display widow as to the number of hits that have occurred and the total number of targets that have been displayed.

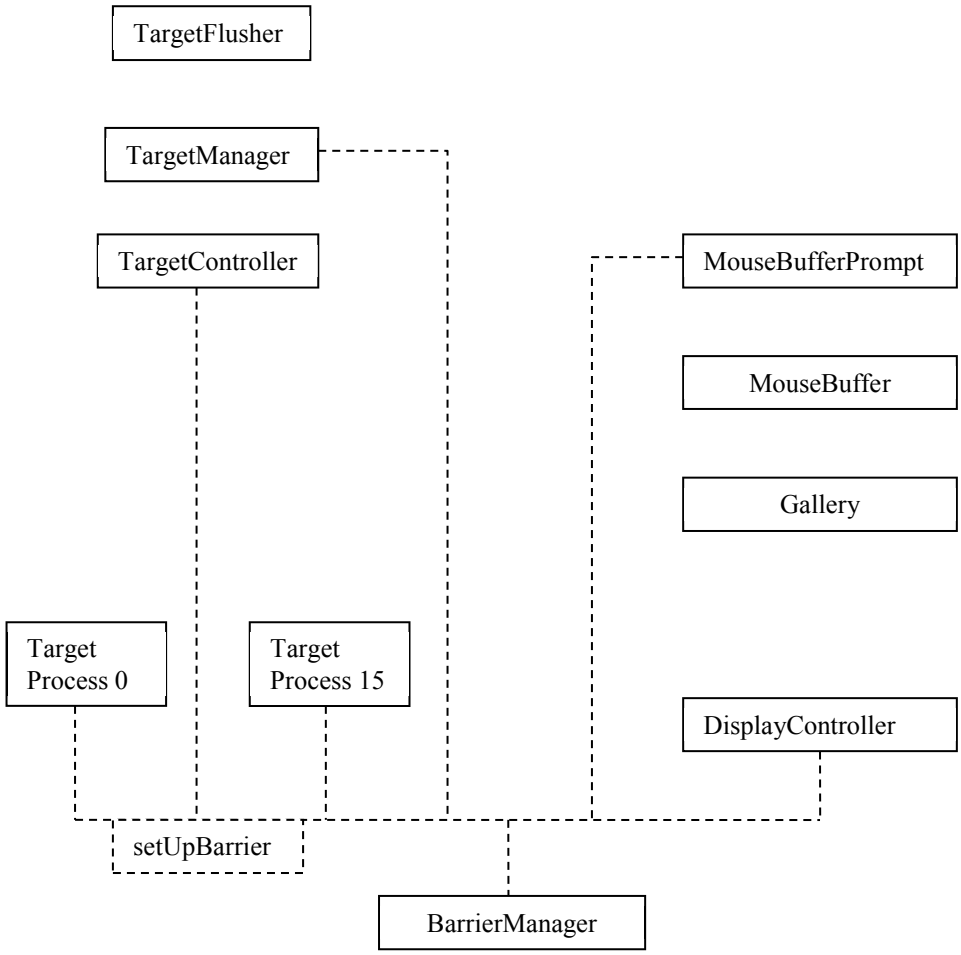


Figure 14-2 System At Setup Barrier Synchronisation

Figure 14-3 shows the system at the `initBarrier` synchronisation, which is the point at which those targets that are executing have initialised themselves and the associated display window is showing the targets. Prior to the `initBarrier` the only process that can execute is `TargetController`. The `TargetController` requests the `TargetManager` to flush the next `Bucket`; a request that is passed onto the `TargetFlusher` process. The `TargetFlusher` accesses the `Bucket`s in sequence until it finds a non-empty one. It then initialises the `initBarrier` with the number of `TargetProcesse`s. It returns this number to the `TargetManager` and then flushes the `TargetProcesse`s, which start running. The `TargetManager` then determines which of the `TargetProcesse`s has been started by waiting for a communication from each of them informing it of the identity of the running targets. These identities are then formed into a `List`, which is then communicated to both the `TargetController` and `DisplayController` processes.

The TargetController can now construct a `ChannelOutputList` that will be subsequently used to communicate the location where mouse presses occur to each of the `TargetProcesse`s. Similarly, the `DisplayController` can modify the display window to show the running targets.

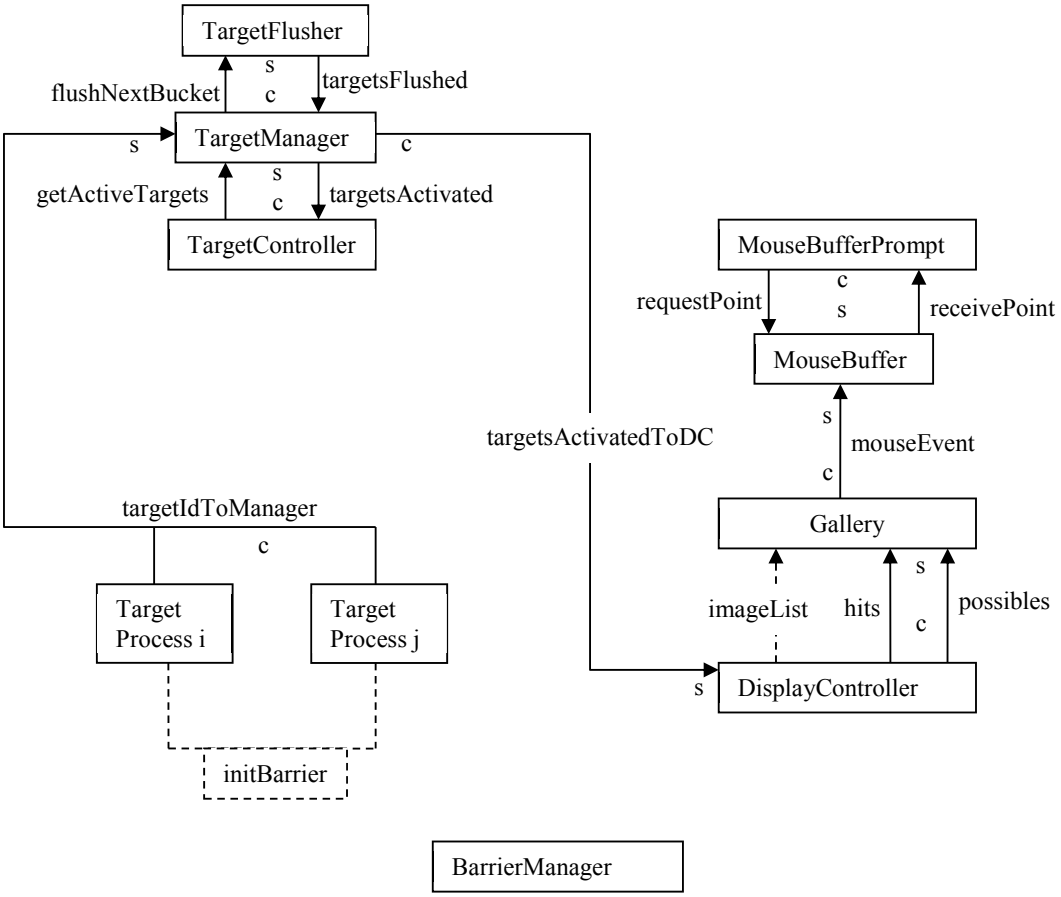


Figure 14-3 System At the Initialise Barrier Synchronisation

The `MouseBufferPrompt` and `MouseBuffer` have a design similar to that used previously in the manipulation of a queue (Chapter 6.2) and event handling (Chapter 11.2). `MouseBuffer` only accepts a request from `MouseBufferPrompt` when it has already received an event on its `mouseEvent` channel. The `Gallery` process is responsible both for the `ActiveCanvas` upon which the targets are displayed and the detection and communication of mouse click events. At this stage the `MouseBufferPrompt` process has no channel on which it can output points but that is not required until the system progresses to the next, `goBarrier` phase.

The `goBarrier` is simply required to ensure that all the running `TargetProcesse`s, the `TargetController` and `DisplayController` have reached a state whereby the system can start execution from a known state. As such this phase does not require any channel communication as shown in Figure 14-4. Once these processes have synchronised the system enters the normal running state of the system with some of the `TargetProcesse`s executing.

Each of the `Barrier`s used so far are of the simple variety because the number of processes that require synchronising can be predetermined and there is no need for any of these `Barrier`s to interact with a possible communication or timer in an alternative. The communications are all required to have completed before the processes can reach the synchronisation point. The remaining `Barrier`s are of the `AltingBarrier` variety because the requirement to synchronise can happen at the same time as a timer alarm or communication occurs.

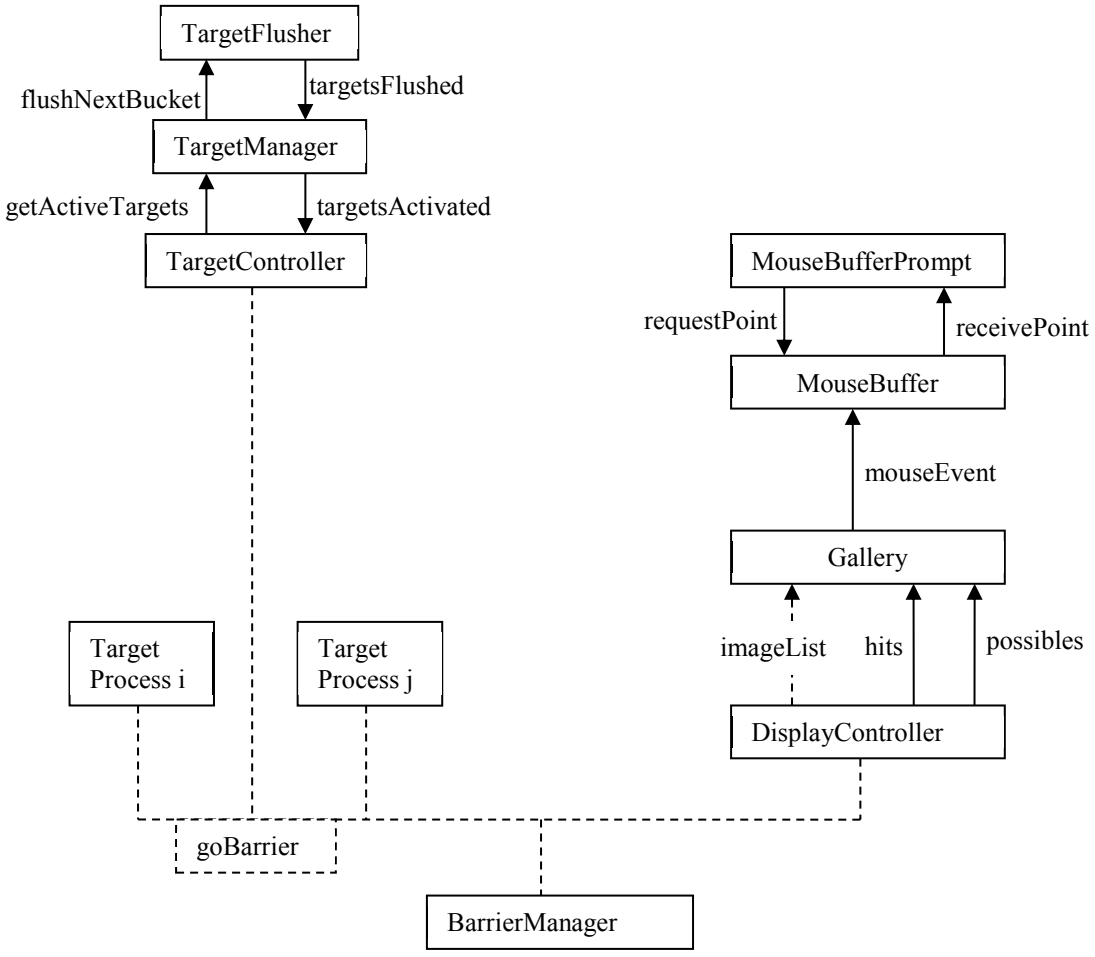


Figure 14-4 System At the Go Barrier Synchronisation

Figure 14-5 shows the system structure when the TargetProcesses are waiting for mouse clicks to determine whether or not they have been hit. The figure also shows the client-server analysis appropriate to this phase of the system's operation.

Initial, cursory inspection, would seem to suggest that there a client-server loop has been created. However, it can be seen that the `MouseBuffer` is a pure server and therefore ensures that no loop is formed. Furthermore, the `Gallery` process provides a user interface capability that has some unusual properties. Any incoming communication is always fully acted upon within the process and is not transmitted further. Thus for its inputs the `Gallery` acts as a pure server. For any mouse events that it might generate, the Gallery acts as a pure client provided any event channels are communicated by a channel that utilises an overwriting buffer. This requirement is expounded further in the JCSP documentation and was discussed in Chapter 11.2.3.

The operation of a `TargetProcess` is specified as follows. After synchronising on the `goBarrier` it calculates its own random alarm time, which then forms part of an alternative that comprises the alarm and channel communications on its `mousePoints` channel. This alternative is looped around until either the alarm time occurs or the target is hit. In either case the target is no longer active. Another alternative is then entered that comprises communications on its `mousePoints` channel or the `timeAndHitBarrier`. Even though a target is inactive other targets may still not yet have timed out and thus mouse clicks will still be received. The `timeAndHitBarrier` determines when either all the targets have been hit or they have all timed out or some combination of these situations has occurred. It also has the effect of breaking the connection between `TargetController` and `MouseBufferPrompt` until the next set of targets are initialised. To ensure this does not cause a problem the channel `pointsToTC` uses an `OverWriteOldestBuffer` data store.

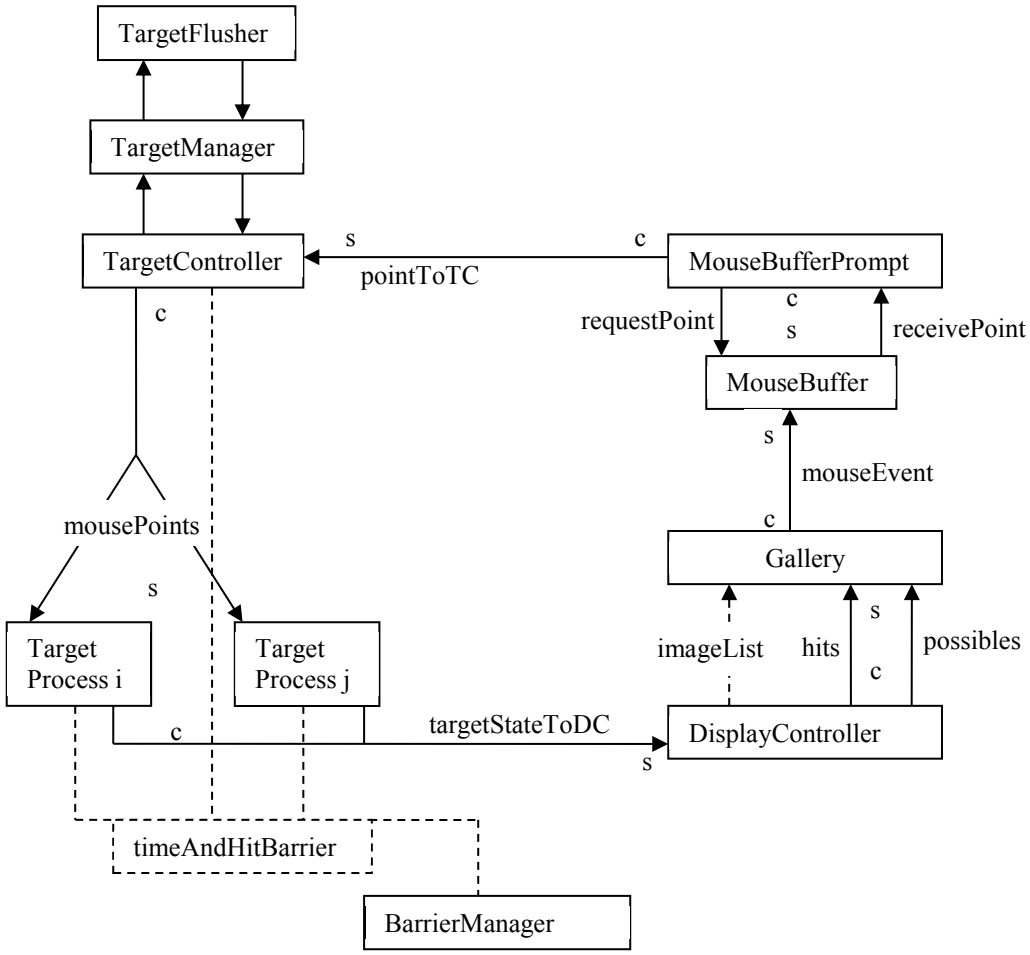


Figure 14-5 System Running Awaiting timeAndHitBarrier

When the state of a target changes (timed out or hit) it sends a communication to the `DisplayController` accordingly, which can then update the display maintained by `Gallery` appropriately. `TargetController` receives a `java.awt.Point` from `MouseBufferPrompt` that give the coordinates where the mouse has been pressed. The `TargetController` then outputs this `Point` value to each of the `TargetProcesses` in parallel using the `ChannelOutputList mousePoints`. Once all the targets have either been hit or timed out the `timeAndHitBarrier` synchronises at which point the `TargetProcesses` individually determine into which randomly chosen `Bucket` they are going to fall.

The system then moves on to the final phase of processing shown in Figure 14-6. The `DisplayController` process contains an alternative with guards comprising the `finalBarrier` and the channel `targetStateToDC`. Thus when it is offering the guard `finalBarrier` together with `BarrierManager` the barrier synchronises and the system is able to progress onto another initial phase as described previously. The only process to undertake any substantial processing in the final phase is the `DisplayController` which leaves the final state of the display for a preset constant time, then sets all the targets to black, thereby obliterating them and then waits for another preset constant time. The coding of each of the processes now follows.

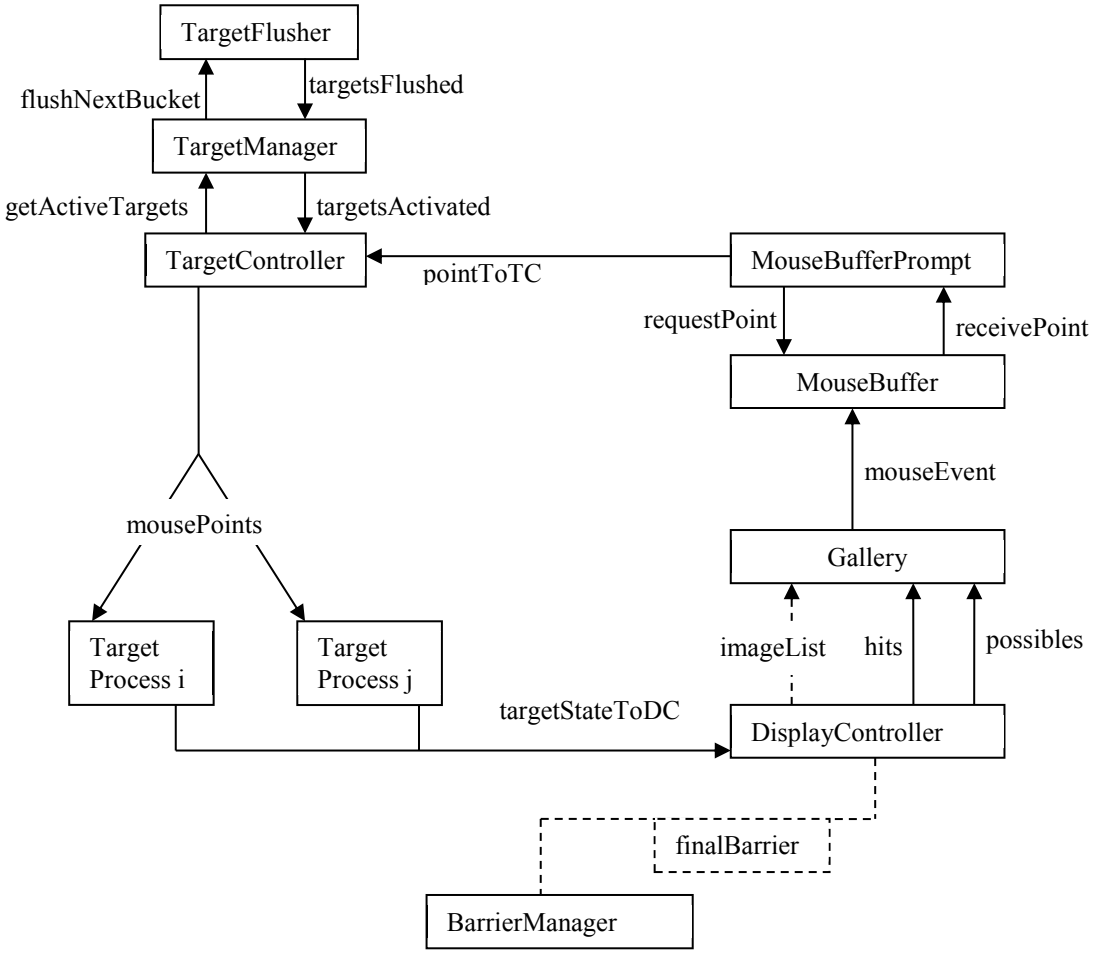


Figure 14-6 System At Final Barrier Synchronisation

14.1 Barrier Manager

The `BarrierManager`, shown in Listing 14-1, simply defines as properties all the barriers in which it participates {12–15}. By definition an `AltingBarrier` must be part of an alternative and thus two `ALT`s are defined {18, 19} in which the particular `AltingBarrier` is the only guard. `BarrierManager` then waits to synchronise on `setUpBarrier` {20}. Thereafter, the process repeatedly synchronises on the `goBarrier`, `timeAndHitBarrier` and `finalBarrier` in sequence {23–25}. A `Barrier` synchronises using the `sync()` method call, whereas synchronisation on an `AltingBarrier` is achieved by calling the `select()` method call of the `ALT` that contains the barrier as a guard. In this case because the guard is the only element in the alternative a simple call of the `select()` method is sufficient, the value returned is of no importance. An alting barrier becomes enabled when all other members of the `AltingBarrier` also `select()` the same alting barrier.

```
10 class BarrierManager implements CSProcess{
11
12   def AltingBarrier timeAndHitBarrier
13   def AltingBarrier finalBarrier
14   def Barrier goBarrier
15   def Barrier setUpBarrier
16
17   void run() {
18     def timeHitAlt = new ALT ([timeAndHitBarrier])
19     def finalAlt = new ALT ([finalBarrier])
20     setUpBarrier.sync()
21
22     while (true){
23       goBarrier.sync()
24       def t = timeHitAlt.select()
25       def f = finalAlt.select()
26     }
27   }
28 }
```

Listing 14-1 Barrier Manager

14.2 Target Controller

Listing 14-2 shows the coding of the `TargetController` process, which is the process that effectively controls the operation of the complete system. The properties of the process are defined {12–20} and these directly implement the channel and barrier structures shown in Figures 14-2 to 14-6.

```
10 class TargetController implements CSProcess {
11
12   def ChannelOutput getActiveTargets
13   def ChannelInput activatedTargets
14   def ChannelInput receivePoint
15   def ChannelOutputList sendPoint
16
17   def Barrier setUpBarrier
18   def Barrier goBarrier
19   def AltingBarrier timeAndHitBarrier
20   def int targets = 16
21
22     void run() {
23       def POINT = 1
24       def BARRIER = 0
25       def controllerAlt = new ALT ( [ timeAndHitBarrier, receivePoint] )
26
27       setUpBarrier.sync()
28       while (true) {
29         getActiveTargets.write(1)
30         def activeTargets = activatedTargets.read()
31         def runningTargets = activeTargets.size
32         def ChannelOutputList sendList = []
33         for ( t in activeTargets) sendList.append(sendPoint[t])
34         def active = true
```

```
35          goBarrier.sync()
36          while (active) {
37            switch ( controllerAlt.priSelect() ) {
38              case BARRIER:
39                active = false
40                break
41              case POINT:
42                def point = receivePoint.read()
43                sendList.write(point)
44                break
45            } // end switch
46          } // end while active
47        } // end while true
48    } // end run
49 }
```

Listing 14-2 Target Controller

Within the `run` method some constants used to identify guards are defined {23, 24} of an alternative {25}. The zero'th guard of the alternative `controllerAlt` is the `AltingBarrier timeAndHitBarrier` and as such is incorporated into an `ALT` like any other guard. The process then waits for all the other enrolled processes to synchronise on `setUpBarrier` {27} before continuing with the unending loop {28–47} that is the main body of the process.

The first action of the process is to send a signal {29} to the `TargetManager` process using the channel `getActiveTargets`. This is the first part of a client-server request and response pair of communications, the second of which is the receipt of a list of the `targetId`s of the `activeTargets` from the channel `activatedTargets` {30}. The `activeTargets` list is then used to create {33} a subset of the `ChannelOutputList` property `sendPoint` {15} in another `ChannelOutputList sendList`, which is used subsequently to communicate with each of the `TargetProcesse`s. The Boolean property `active` is then defined {34} and will be used to control the subsequent operation of the process. The process now waits to synchronise on the `goBarrier` {35}. Prior to the `goBarrier` synchronisation all the `TargetProcesse`s will have synchronised on the `initBarrier` but that is of no concern to the `TargetController` process.

The `goBarrier` is used to synchronise the operation of all the targets in the running `TargetProcesse`s, the `BarrierManager` and the `DisplayController` as well as `TargetController`. The synchronisation enables each of these processes to run in that part of the system which allows users to move their mouse over the active targets and to try and hit each of them, by means of a mouse press, before each target times out. Thus the only actions that can occur are either, a mouse press occurs, or all the targets have either been hit or timed out. The mouse press manifests itself as the input of a `Point` on the `receivePoint` channel {42}. The value of `point` is then communicated, in parallel {43}, to all the members of `sendList` to each of the running `TargetProcesse`s. (A `write` on a `ChannelOutputList` causes the writing of the method call parameter to all the channels in the list in parallel). If the barrier guard is `selected` then the loop terminates as soon as all the other processes on the `timeAndHitBarrier` have been selected {38}. The value of `active` is set `false` {39}which causes the inner while loop to terminate {36} ready for the process to cycle again round the outer non-terminating while loop {28}.

14.3 Target Manager

Listing 14-3 shows the coding of the `TargetManager` process. Its properties are defined {12–18}. The process does not have anything to do prior to the `setUpBarrier` synchronisation {21}. Its body comprises a non-terminating loop {22–34}. Initially, it reads the signal from `TargetController` on its `getActiveTargets` channel {24}, which causes the writing of yet a further signal to the `TargetFlusher` process on the `flushNextBucket` channel {25}. This is also the first part of the client-server communication pattern between `TargetManager` and `TargetFlusher`. The corresponding response is `read` from the `targetsFlushed` channel, which is the number of `TargetProcesse`s that have been initialised into the variable `targetsRunning` {26}. The next phase {27–30} is to read from each of the initialised `TargetProcesse`s their identity on the `targetIdFromTarget` channel and append it to the `targetList` {28}. This list is then written to the `TargetController` process {31} using the `activatedTargets` channel, thereby completing the client-server interaction between `TargetManager` and `TargetController`. Finally, the list of initialised targets is written to the `DisplayController` using the channel `activatedTargetsToDC` {32}. These two communications allow the receiving process to complete their initialisation prior to further operation. The process then cycles waiting to read the next group of actived targets {24}, which cannot be undertaken until the next bucket is flushed.

```
10 class TargetManager implements CSProcess {
11
12   def ChannelInput targetIdFromTarget
13   def ChannelInput getActiveTargets
14   def ChannelOutput activatedTargets
15   def ChannelOutput activatedTargetsToDC
16   def ChannelInput targetsFlushed
17   def ChannelOutput flushNextBucket
18   def Barrier setUpBarrier
19
20   void run() {
21     setUpBarrier.sync()
22     while (true) {
23       def targetList = [ ]
24       getActiveTargets.read()
25       flushNextBucket.write(1)
26       def targetsRunning = targetsFlushed.read()
27       while (targetsRunning > 0) {
28           targetList << targetIdFromTarget.read()
29           targetsRunning = targetsRunning - 1
30       } // end of while targetsRunning
31       activatedTargets.write(targetList)
32       activatedTargetsToDC.write(targetList)
33     } // end of while true
34   }
34 }
```

Listing 14-3 Target Manager

14.4 Target Flusher

The role of the `TargetFlusher` process, shown in Listing 14-4, is to manage the `Buckets` into which the `TargetProcesses` fall.

```
10 class TargetFlusher implements CSProcess {
11
12   def buckets
13   def ChannelOutput targetsFlushed
14   def ChannelInput flushNextBucket
15   def Barrier initBarrier
16
17   void run() {
18     def nBuckets = buckets.size()
19     def currentBucket = 0
20     def targetsInBucket = 0
21     while (true) {
22       flushNextBucket.read()
23       targetsInBucket = buckets[currentBucket].holding()
```

```
24        while ( targetsInBucket == 0) {
25          currentBucket = (currentBucket + 1) % nBuckets
26          targetsInBucket = buckets[currentBucket].holding()
27        } // end of while targetsInBucket
28        initBarrier.reset( targetsInBucket)
29        targetsFlushed.write(targetsInBucket)
30        buckets[currentBucket].flush()
31        currentBucket = (currentBucket + 1) % nBuckets
32      } // end of while true
33    }
34 }
```

Listing 14-4 Target Flusher

The process also completes the client-server interaction with the `TargetManager` process. Its properties are defined {12–15}. Some variables are initialised {18–20} in the first part of the `run` method. The main loop of the process {21–32} initially reads the signal {22} that causes it to start the initialisation of some `TargetProcesses`. The number of `TargetProcesses` in the `currentBucket` is determined by means of a call of the `holding()` method {23}. The next piece of coding {24–27} ensures that the number of `TargetProcesses` that are `flushed` is greater then zero.

At this stage `initBarrier` can be set to the number of `targetsInBucket` {28} by means of a call to the `reset` method. The number of `targetsInBucket` can now be written to the `TargetManager` process {29}. Now the `TargetProcesses` contained in the `currentBucket` can be `flush`ed {30} and therefore start running. Finally, the value of `currentBucket` can be incremented subject to its value staying within zero to the number of Buckets, `nBuckets` {31}.

14.5 Display Controller

The `DisplayController` process is shown in Listings 14-5 to 14-8 and manages the interaction between the `TargetProcesses` and the user interface provided by the `Gallery` process, described in the next section.

The TargetProcesses communicate with the `DisplayController` by means of the channel `stateChange` {11}, which is the 'one' end of an `any2one` channel. The channel `activeTargets` {12} is used to input the list of running targets during the initial phase of a cycle. The `displayList` property {14} provides the connection between this process and the `ActiveCanvas` contained in the `Gallery` process. The channels `hitsToGallery` and `possiblesToGallery` {15, 16} are used to send values to the `ActiveLabels` in the `Gallery` process that display the number of targets that have been hit and the total number of targets displayed. Finally, the barriers upon which `DisplayController` synchronises are defined {18–20}.

```
10 class DisplayController implements CSProcess {
11   def ChannelInput stateChange
12   def ChannelInput activeTargets
13
14   def DisplayList displayList
15   def ChannelOutput hitsToGallery
16   def ChannelOutput possiblesToGallery
17
18   def Barrier setUpBarrier
19   def Barrier goBarrier
20   def AltingBarrier finalBarrier
21
```

Listing 14-5 Display Controller Properties

Listing 14-6 gives the array of `GraphicsCommands` and list of values used to change the `displayList`. These are not shown complete, but are those parts that relate to the first and last. The array `targetGraphics` is used to initially create the `displayList`. Each of the elements of the list `targetColour` comprises the colour of the target and the element of `targetGraphics` that has to be changed in order to display the target. The first two elements of targetGraphics {25, 26} have the effect of completely 'blacking' out the display canvas prior to its repainting within the `Canvas` thread.

```
22  void run() {
23
24    def GraphicsCommand [] targetGraphics = new GraphicsCommand [ 34 ]
25    targetGraphics[0] = new GraphicsCommand.SetColor (Color.BLACK)
26    targetGraphics[1] = new GraphicsCommand.FillRect (0, 0, 450, 450)
27    targetGraphics[2] = new GraphicsCommand.SetColor (Color.BLACK)
28    targetGraphics[3] = new GraphicsCommand.FillRect (10, 10, 100, 100)
29    targetGraphics[4] = new GraphicsCommand.SetColor (Color.BLACK)
30    targetGraphics[5] = new GraphicsCommand.FillRect (120, 10, 100, 100)
31    targetGraphics[6] = new GraphicsCommand.SetColor (Color.BLACK)
32    targetGraphics[7] = new GraphicsCommand.FillRect (230, 10, 100, 100)
33 …
34    targetGraphics[30] = new GraphicsCommand.SetColor (Color.BLACK)
35    targetGraphics[31] = new GraphicsCommand.FillRect (230, 340, 100, 100)
36    targetGraphics[32] = new GraphicsCommand.SetColor (Color.BLACK)
37    targetGraphics[33] = new GraphicsCommand.FillRect (340, 340, 100, 100)
38
39    def targetColour = [
40      [new GraphicsCommand.SetColor (Color.RED), 2],
41      [new GraphicsCommand.SetColor (Color.GREEN), 4],
42      [new GraphicsCommand.SetColor (Color.YELLOW), 6],
43      [new GraphicsCommand.SetColor (Color.BLUE), 8],
44 …
45      [new GraphicsCommand.SetColor (Color.MAGENTA), 30],
46      [new GraphicsCommand.SetColor (Color.ORANGE), 32]
47                          ]
```

Listing 14-6 Graphics definitions

The `run` method has some further properties that are shown in Listing 14-7, which include the constants {48, 49} used to identify the selected alternative defined as `controllerAlt` {52}. The constants {54–56} define the `GraphicsCommand` that can be used to colour a square as indicated by their name. Finally, variables that tally the number of hits and possible hits are defined {58, 59} together with a timer {60} that is used to control the time the display stays static at the end of a cycle.

```
48      def CHANGE = 1
49      def BARRIER = 0
50      def TIMED_OUT = 0
51      def HIT = 1
52      def controllerAlt = new ALT ( [ finalBarrier, stateChange ] )
53
54      def whiteSquare = new GraphicsCommand.SetColor(Color.WHITE)
55      def blackSquare = new GraphicsCommand.SetColor(Color.BLACK)
56      def graySquare = new GraphicsCommand.SetColor(Color.GRAY)
57
58      def totalHits = 0
59      def possibleTargets = 0
60      def timer = new CSTimer()
```

Listing 14-7 Other Run Method Properties

The body of the `run` method is shown in Listing 14-8. Prior to the `setUpBarrier` synchronisation {64} the `displayList` is initialised by a call to the `set` method {61} and the initial, zero, values of `totalHits` and `possibleHits` are written to the `Gallery` {62, 63}.

The never ending loop of the `run` method is then entered {66–99}, which comprises some initialisation prior to the `goBarrier` synchronisation {67–73} followed by the active part of the cycle {74–93} until the `finalBarrier` is selected {89–90}.

```
61  displayList.set (targetGraphics)
62  hitsToGallery.write (" " + totalHits)
63  possiblesToGallery.write ( " " + possibleTargets )
64  setUpBarrier.sync()
65
66  while (true) {
67    def active = true
68    def runningTargets = activeTargets.read()
69    possibleTargets = possibleTargets + runningTargets.size
70    possiblesToGallery.write ( " " + possibleTargets )
71    for ( t in runningTargets)
72      displayList.change ( targetColour[t][0], targetColour[t][1])
73    goBarrier.sync()
74    while (active) {
75      switch (controllerAlt.priSelect()) {
```

```
76          case CHANGE:
77              def modification = stateChange.read() // [ tId, state ]
78              switch ( modification[1]) {
79                case HIT:
80                  displayList.change ( whiteSquare, targetColour[modification[0]][1])
81                  totalHits = totalHits + 1
82                  hitsToGallery.write (" " + totalHits)
83                 break
84                case TIMED_OUT:
85                  displayList.change ( graySquare, targetColour[modification[0]][1])
86                break
87              } // end switch modification
88              break
89          case BARRIER:
90              active = false
91              break
92          } // end switch controllerAlt
93       } // end of while active
94       timer.sleep(1500)
95       for ( tId in runningTargets )
96           displayList.change ( blackSquare, targetColour[tId][1])
97       timer.sleep ( 500)
98     } // end while true
99    }
100 }
```

Listing 14 – 8 Run Method Definition

The process `DisplayController` is initialised by reading the identities of the running targets into the list `runningTargets` from `TargetManager` using the channel `activeTargets` {68}. The `size` of this list is then used to update the total number of possible targets in the `Gallery` {69–70}. The members of the list are then used to `change` the `displayList`, which cause the targets to appear in the `Gallery` {71–72}. The process then synchronises on the `goBarrier` {73}.

The process remains `active` {74} until the `finalBarrier` is selected {89–90}. It should be noted that the order of the guards in `controllerAlt` is important, with priority given to inputs from the `TargetProcesse`s, so that all changes to the targets are completed before the `finalBarrier` is `selected`. While the process is `active`, communications from the running `TargetProcesse`s are `read` from the channel `stateChange` {77} which are used to modify the state of the targets in the `Gallery` by `chang`ing the `displayList`. The input from a `TargetProcess` is in the form of a list comprising the identity of the target and the state to which it should be changed. Two state changes are possible indicated by `HIT`, when the target's image is changed to white {80} and the number of targets hit is also updated {81–82} and `TIMED_OUT` when the square is coloured grey {85}.

Once the `finalBarrier` has been `selected` {89, 90} the process `sleeps` for 1.5 seconds {94} to allow the user to determine the final state of that cycle. The running targets, which are now all coloured either white or grey are returned to the colour black {95–96}. The process `sleeps` for a further 0.5 seconds {97}, to provide the user a break between cycles of the system. It then resumes the main loop of the process, which is initiated by reading the identities of the targets that have been flushed from the next Bucket.

14.6 Gallery

The `Gallery` process shown in Listing 14-9 is similar to other user interface processes that have been discussed previously. The only aspect of particular note is that a mouse event channel {15} is added to the `ActiveCanvas` {39}. There is no need for the programmer to add anything further in terms of listener of event handling methods. Any mouse event is communicated on the `mouseEvent` channel to the `MouseBuffer` process. The components of the interface can be seen, by observation, to produce that shown in Figure 14-1.

```
10 class Gallery implements CSProcess{
11
12   def ActiveCanvas targetCanvas
13   def ChannelInput hitsFromGallery
14   def ChannelInput possiblesFromGallery
15   def ChannelOutput mouseEvent
16   def canvasSize = 450
17
18   void run() {
19     def root = new ActiveClosingFrame ("Hand-Eye Co-ordination Test")
20     def mainFrame = root.getActiveFrame()
21     def m1 = new Label ("You Have Hit")
22     def m2 = new Label ("Out Of")
23     def hitLabel = new ActiveLabel (hitsFromGallery)
24     def possLabel = new ActiveLabel (possiblesFromGallery)
25     m1.setAlignment( Label.CENTER)
26     m2.setAlignment( Label.CENTER)
27     hitLabel.setAlignment( Label.CENTER)
28     possLabel.setAlignment( Label.CENTER)
29     m1.setFont(new Font("sans-serif", Font.BOLD, 14))
30     m2.setFont(new Font("sans-serif", Font.BOLD, 14))
31     hitLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
32     possLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
33     def message = new Container()
34     message.setLayout ( new GridLayout ( 1, 4 ) )
35     message.add (m1)
36     message.add (hitLabel)
37     message.add (m2)
38     message.add (possLabel)
39     targetCanvas.addMouseEventChannel ( mouseEvent )
```

```
40      mainFrame.setLayout( new BorderLayout() )
41      targetCanvas.setSize (canvasSize, canvasSize)
42      mainFrame.add (targetCanvas, BorderLayout.CENTER)
43      mainFrame.add (message, BorderLayout.SOUTH)
44      mainFrame.pack()
45      mainFrame.setVisible ( true )
46      def network = [ root, targetCanvas, hitLabel, possLabel ]
47      new PAR (network).run()
48    }
49 }
```

Listing 14-9 Gallery Process

14.7 Mouse Buffer

The MouseBuffer, shown in Listing 14-10 process reads mouse events on its mouseEvent channel {12}. Only when the event is a MOUSE_PRESSED event does it store the location of the click {35} in the variable point. At this stage it modifies {34} the pre-condition on the process' alternative, mouseBufferAlt so as to be able to accept requests for a point {26}, which can then be transferred to the MouseBufferPrompt process {28}, after which the pre-condition is again modified {29} so as not to accept further prompt requests until another mouse click point has been received. This mechanism was used previously in the Queue and Event Handling Systems and is an idiom or pattern used to manage requests for external non-deterministic events. In this case we note that the mouseEvent channel is always available to read events and thus does not block the Gallery process with its implicit threads that are used to implement events and a canvas. This is further demonstrated by the mouseEvent channel having a data store associated with it that enables the overwriting of the oldest member of the associated buffer (see 14.10).

```
10  class MouseBuffer implements CSProcess{
11
12     def ChannelInput mouseEvent
13     def ChannelInput getClick
14     def ChannelOutput sendPoint
15
16     void run() {
17       def mouseBufferAlt = new ALT ( [ getClick, mouseEvent ] )
18       def preCon = new boolean [2]
19       def EVENT = 1
20       def GET = 0
21       preCon[EVENT]= true
22       preCon[GET] = false
23       def point
24       while (true) {
25         switch (mouseBufferAlt.select(preCon)) {
26           case GET:
27             getClick.read()
28             sendPoint.write(point)
29             preCon[GET] = false
30             break
31           case EVENT:
32             def mEvent = mouseEvent.read()
33             if ( mEvent.getID() == MouseEvent.MOUSE_PRESSED) {
34               preCon[GET] = true
35               point = mEvent.getPoint()
36             }
37             break
38         }
39       }
40     }
41  }
```

Listing 14-10 Mouse Buffer Process

14.8 Mouse Buffer Prompt

The `MouseBufferPrompt` process shown in Listing 14-11, simply writes a request to the `getPoint` channel {20} and then waits to read a `point` on the `receivePoint` channel {21} which it then `writes` to the `TargetController` process on the `returnPoint` channel {22}. The combination of `MouseBufferPrompt` and `MouseBuffer` ensures that the `MouseBuffer` process is a pure server in a client-server analysis and also has the effect of decoupling the generation of mouse events in the `Gallery` from the process in which they are consumed, `TargetController`. Furthermore, any delay in reading a `point` by the `TargetController` does not cause a delay that might cause the blocking of the implicit event handling thread of `Gallery`.

```
10  class MouseBufferPrompt implements CSProcess{
11
12     def ChannelOutput returnPoint
13     def ChannelOutput getPoint
14     def ChannelInput receivePoint
15     def Barrier setUpBarrier
16
17     void run () {
18       setUpBarrier.sync()
19       while (true) {
20         getPoint.write( 1 )
21         def point = receivePoint.read()
22         returnPoint.write( point )
23       }
24     }
25  }
```

Listing 14-11 Mouse Buffer Prompt Process

14.9 Target Process

The `TargetProcess` is shown in Listings 14-12 to 14-14. The channel `targetRunning` {12} is used by `TargetProcess` to inform the `TargetManager` process that it has been flushed from a `Bucket` and has been made active. The channel `stateToDC` {13} is used to inform the `DisplayController` of any change in state of this target that is, either hit or timed-out. The channel `mousePoint` {14} is used to input the `java.awt.Point` at which the mouse has been pressed. The process is a member of the `setUp`, `init`, `go` and `timeAndHit` barriers {15–18}. It also requires access to the array of `buckets` {19}. The property `targetId` {20} is a unique integer identifying the instance of `TargetProcess` and the values `x` {21} and `y` {22} are the pixel co-ordinates of the upper left corner of the target in the display window. The property `delay` {23} specifies the minimum period for which the target will be displayed before it times out. The target will be visible for a random time between `delay` and twice `delay`. The method `within` {25–33} determines if a `java.awt.Point p` is within the target area. All targets are square with a side of 100 pixels.

```
10  class TargetProcess implements CSProcess {
11
12     def ChannelOutput targetRunning
13     def ChannelOutput stateToDC
14     def ChannelInput mousePoint
15     def Barrier setUpBarrier
16     def Barrier initBarrier
17     def Barrier goBarrier
18     def AltingBarrier timeAndHitBarrier
19     def buckets
20     def int targetId
21     def int x
22     def int y
23     def delay = 2000
24
25     def boolean within ( Point p, int x, int y) {
26       def maxX = x + 100
27       def maxY = y + 100
28       if ( p.x < x ) return false
29       if ( p.y < y ) return false
30       if ( p.x > maxX ) return false
31       if ( p.y > maxY ) return false
32       return true
33     }
34
```

Listing 14-12 The Properties and Within Method of target process

The first part of the `run` method is executed during the setup phase of the system and is only executed once, Listing 14-13. A Random number generator `rng` {36} is defined and then used to specify the initial bucket, `bucketId` {38, 39} into which the `TargetProcess` will subsequently fall. Initially all `TargetProcesses` will fall into a bucket in the first half of the array of buckets. A `timer` and some constants are also defined {37, 40–44}.

Two alternatives are then defined. The alternative `preTimeOutAlt` {46} is used prior to the `TargetProcess` being timed out and `postTimeOutAlt` {47} is used once a time out has occurred or the target has been hit. The latter alternative includes the `AltingBarrier` `timeAndHitBarrier`. The operation of such an `AltingBarrier` is straightforward. It must appear as a guard in an alternative. Whenever any select method on the alternative is called a check is made to determine whether all the other members of the `AltingBarrier` have also requested and are waiting on such a select. If they have, then, the `AltingBarrier` as a whole can be selected. If one of the members of an `AltingBarrier` accepts another guard in such an alternative then the `AltingBarrier` cannot be selected. Thus it is possible for a process to offer an `AltingBarrier` guard and then withdraw from that guard if the dynamics of the system cause that to happen.

The `TargetProcess` now resigns from `timeAndHitBarrier` {49}, which at first sight may seem perverse. All `TargetProcesses` are initially enrolled on this barrier. However we only want running targets to be counted as part of the barrier so we must first resign from the barrier and then `enroll` only when the `TargetProcess` is executed.

The mechanism of `enroll` and `resign` can be applied to all types of barrier. A process that `enrolls` on a barrier can now call the sync method (Barrier) or be a guard in an alternative and thus can be selected (`AltingBarrier`). Similarly a process can resign which means that the process is no longer part of the barrier. In the case of a `Barrier` resignation it also implies that if this is the last process to synchronise on the `Barrier` then this is equivalent to all the processes having synchronised. A process cannot resign if it is not enrolled. In the case of `AltingBarriers` this enrolment and resignation has to be undertaken with care as no process can be running and selecting the barrier onto which it is intended to either enrol or resign another process from. The associated documentation for JCSP specifies this requirement more fully.

The `TargetProcesses` now synchronise on the `setUpBarrier` {50} and when this is achieved they then `fallInto` the bucket with subscript `bucketId` {51}. This has the effect of stopping the process. It will only be rescheduled when the `TargetFlusher` process causes the bucket into which the process has fallen is `flushed` {Listing 14-4, 30}.

```
35    void run() {
36      def rng = new Random()
37      def timer = new CSTimer()
```

```
38      def int range = buckets.size() / 2
39      def bucketId = rng.nextInt( range )
40      def POINT= 1
41      def TIMER = 0
42      def BARRIER = 0
43      def TIMED_OUT = 0
44      def HIT = 1
45
46      def preTimeOutAlt = new ALT ([ timer, mousePoint ])
47      def postTimeOutAlt = new ALT ([ timeAndHitBarrier, mousePoint ])
48
49      timeAndHitBarrier.resign()
50      setUpBarrier.sync()
51      buckets[bucketId].fallInto()
```

Listing 14-13 Target process: The Setup Phase of Run

The remainder of the `run` method, Listing 14-14, only gets executed when the process has been `flush`ed. It comprises a never ending loop {52–94}, which as its final statement {93} causes itself to fall into another bucket, prior to returning to the start of the loop. The loop itself has three phases comprising the phases managed by `initBarrier` and then that managed by the `goBarrier` before finally running until either the target is hit or times out which is managed by the `timeAndHitBarrier`.

In the initial phase, the process `enroll`s on the `timeAndHitBarrier` {53} and also the `goBarrier` {54}. Enrolling on the `timeAndHitBarrier` causes no problem because at this stage no process can be selecting a guard from an alternative in which `timeAndHitBarrier` is involved. Similarly, enrolling on the `goBarrier` is an operation that can be undertaken dynamically because it is a `Barrier`. The running process now writes its unique identity, `targetId` to its `targetRunning` channel {55}. This communication means that the `TargetManager` now can determine {Listing 14-4, 27–30} which targets are active. It then synchronises on the `initBarrier` {56}. The number of running `TargetProcesse`s associated with the `initBarrier` is specified by `TargetFlusher` {Listing 14-3, 29} at a time when none of these processes can be running because they have yet to be flushed. Only the running `TargetProcesse`s are allowed to access the `initBarrier` and thus once the `initBarrier` has synchronised we know that all the `TargetProcesse`s are in the same state and that any dependent processes such as `DisplayController` will be able to complete any further initialisation prior to the `goBarrier` synchronisation. The Boolean `running` is initialised {57}, which will be used subsequently to control the operation of the process. Similarly, the variable `resultList` is initialised {58} and will be used to indicate the change of state that will occur in the target. The process can now synchronise on the `goBarrier` by `resign`ing from it {59}. The only permanent members of the `goBarrier` are `BarrierManager`, `TargetController` and `DisplayController`, all of which simply call the method `sync()` on the barrier . The `goBarrier` is augmented by the active `TargetProcesse`s to ensure that all the processes are in a state that will be suitable for the whole system to become active.

Once the process has synchronised on the `goBarrier` it determines the time for which the target will be displayed and sets the `timer` alarm {60} which is a guard in the `preTimeOutAlt` (46}. Prior to the alarm occurring only two things can occur, either the `TIMER` alarm does happen {63} or a mouse click `POINT` is received {68}. In the former case, the value `TIMED_OUT` can be appended to the `resultList` {65} and this list can be written to the `DisplayController` using the channel `stateToDC` {66}. Otherwise, an input can be processed {69} which, if it is `within` the target area {70} causes the value `HIT` to be appended to the `resultList` {72} and as before written to the `DisplayController` process {73}. If the point is not `within` the target then the loop repeats until one of the above cases occurs. Once this happens the value of `running` is set `false` {64} and the loop {61–79} terminates.

The process now has take account of the case where other targets are still running; awaiting a time out or a hit, and so mouse clicks and their associated point data will still be received by the `TargetProcess`. Such point data can be ignored {87–89} and only when all the `TargetProcesses` are `selecting` the `timeAndHitBarrier`, together with `TargetController` and `BarrierManager` processes can the `awaiting` loop {81–91} terminate. When this occurs the process `resigns` from the `timeAndHitBarrier` and causes the loop to exit.

The TargetProcess can now prepare itself for falling into another bucket by calculating {92} into which bucket it will fall and then calling the `fallInto` method {93}. The chosen bucket is at least two further on than the current bucket which means that it cannot be flushed in the next iteration of `TargetFlusher`, unless the next bucket happens to be empty.

```
52      while (true) {
53        timeAndHitBarrier.enroll()
54        goBarrier.enroll()
55        targetRunning.write(targetId)
56        initBarrier.sync() //ensures all targets have initialised
57        def running = true
58        def resultList = [targetId]
59        goBarrier.resign()
60        timer.setAlarm( timer.read() + delay + rng.nextInt(delay) )
61        while ( running ) {
62          switch (preTimeOutAlt.priSelect() ) {
63            case TIMER:
64              running = false
65              resultList << TIMED_OUT
66              stateToDC.write(resultList)
67              break
68            case POINT:
69              def point = mousePoint.read()
70              if ( within(point, x, y) ) {
71                running = false
72                resultList << HIT
73                stateToDC.write(resultList)
74              }
75              else {
76              }
77              break
78          }
79        } // end while running
80        def awaiting = true
81        while (awaiting) {
82          switch (postTimeOutAlt.priSelect() ) {
83            case BARRIER:
84              awaiting = false
85              timeAndHitBarrier.resign()
86              break
87            case POINT:
88              mousePoint.read()
89              break
90          }
91        } // end while awaiting
92        bucketId = ( bucketId + 2 + rng.nextInt(8) ) % buckets.size()
```

```
93        buckets[bucketId].fallInto()
94      }// end while true
95    }
96 }
```

Listing 14-14 Target Process: The Active Phase of the Run Method

14.10 Running the System

Listing 14-15 gives the declarations of the channels, barriers and other data required to create the network according to the process network diagrams given in Figures 14-2 to 14-6 and as such are not particularly noteworthy apart from those described below. The `Barrier`s are defined with the required number of processes. Thus `setUpBarrier` {18} is defined with the number of `targets` plus five for the other processes that use this barrier, see Figure 14-2. The `initBarrier` {19} is defined with no members because only the running `TargetProcesse`s use this barrier and the number is `reset` explicitly in `TargetFlusher`, see Figure 14-3. Finally, the `goBarrier` {20} is defined has having three members, which are the permanently attached processes as shown in Figure 14-4.

The `AltingBarrier`s are defined as an array, with sufficient members such that every process that access them may have a so-called 'front-end'. The `finalBarrier` {23} only requires two front-ends because only `BarrierManager` and `DisplayController` participate in this barrier. The barrier `timeAndHitBarrier` {22} requires a front-end for each `TargetProcess`, the `TargetController` and `BarrierManager`. Each process participating in an `AltingBarrier` needs to be allocated its own front-end so that it can access the barrier during an alternative `select()` method call. Recall that as a `TargetProcess` becomes active it specifically `enroll`s on the `timeAndHitBarrier` thereby activating its membership of the barrier and when its turn is complete it `resign`s from it. Thus the number of processes that are members of the `timeAndHitBarrier` is determined dynamically at run time. The `Bucket`s are defined by means of a `create` method call {25} and this could be any sensible number to provide a wide variety of target initiations per cycle, too many buckets and we would get too few running targets to make the challenge interesting!

```
10  def delay = Ask.Int("Target visible period (2000 to 3500)? ", 2000, 3500)
11
12  def targets = 16
13  def targetOrigins = [ [10, 10],[120, 10],[230, 10],[340, 10],
14                        [10, 120],[120, 120],[230, 120],[340, 120],
15                        [10, 230],[120, 230],[230, 230],[340, 230],
16                        [10, 340],[120, 340],[230, 340],[340, 340] ]
17
18  def setUpBarrier = new Barrier(targets + 5)
19  def initBarrier = new Barrier()
20  def goBarrier= new Barrier(3)
21
22  def timeAndHitBarrier = AltingBarrier.create(targets+2)
```

```
23  def finalBarrier = AltingBarrier.create(2)
24
25  def buckets = Bucket.create(targets)
26
27  def mouseEvent = Channel.one2one ( new OverWriteOldestBuffer(20) )
28  def requestPoint = Channel.one2one()
29  def receivePoint = Channel.one2one()
30  def pointToTC = Channel.one2one( new OverWriteOldestBuffer(1) )
31
32  def targetsFlushed = Channel.one2one()
33  def flushNextBucket = Channel.one2one()
34
35  def targetsActivated = Channel.one2one()
36  def targetsActivatedToDC = Channel.one2one()
37  def getActiveTargets = Channel.one2one()
38
39  def hitsToGallery = Channel.one2one()
40  def possiblesToGallery = Channel.one2one()
41
42  def targetIdToManager = Channel.any2one()
43  def targetStateToDC = Channel.any2one()
44
45  def mousePointToTP = Channel.one2oneArray(targets)
```

```
46  def mousePoints = new ChannelOutputList ( mousePointToTP )
47
48  def imageList = new DisplayList()
49  def targetCanvas = new ActiveCanvas ()
50  targetCanvas.setPaintable ( imageList )
51
```

Listing 14-15 Running the System Property Definitions

The `mouseEvent` channel {27} must be defined with a data store of type `OverWriteOldestBuffer` so that the event handling thread associated with the user interface does not block; see the JCSP documentation for `ActiveCanvas`. Similarly the `pointToTC` channel also uses a one place `OverWriteOldestBuffer` {30} so that if mouse clicks are received too quickly the system does not block. Given the normal performance of a PC this is very unlikely to occur as the user time to move the mouse to another target is relatively long.

The channels that connect `TargetController` to the `TargetProcesses` are defined as an array, `mousePointToTP` {45}, the input end of which is passed directly to the `TargetProcess`. The output ends are formed into a `ChannelOutputList`, `mousePoints` {46}, so that they can be written to in parallel by a `write` method call by `TargetController`.

The `DisplayList` and `ActiveCanvas` components are defined {48–50} prior to being passed as properties of the required processes.

Listing 14-16 shows the definition of the `TargetProcesses` and also of `BarrierManager`. The other processes can be found in the accompanying software because they are very similar to the definition of processes in other systems. It is a matter of tying together the property definition in the process and the defined variable in the script that causes the system to execute. The barriers are straightforward but the allocation of a `timeAndHitBarrier` requires that a specific front-end is allocated to each `TargetProcess` {60} and also to `BarrierManager` {70}. The origin co-ordinates of each `TargetProcess` {63, 64} for the associated display is obtained from the list `targetOrigins`.

```
52  def targetList = ( 0 ..< targets ).collect { i ->
53                   return new TargetProcess (
54                       targetRunning: targetIdToManager.out(),
55                       stateToDC: targetStateToDC.out(),
56                       mousePoint: mousePointToTP[i].in(),
57                       setUpBarrier: setUpBarrier,
58                       initBarrier: initBarrier,
59                       goBarrier: goBarrier,
60                       timeAndHitBarrier: timeAndHitBarrier[i],
61                       buckets: buckets,
62                       targetId: i,
```

```
63                      x: targetOrigins[i][0],
64                      y: targetOrigins[i][1],
65                      delay: delay
66                      )
67                  }
68
69  def barrierManager = new BarrierManager (
70                      timeAndHitBarrier: timeAndHitBarrier[targets],
71                      finalBarrier: finalBarrier[0] ,
72                      goBarrier: goBarrier,
73                      setUpBarrier: setUpBarrier
74                      )
75
```

Listing 14-16 Decalring the TargetProcesses and BarrierManager

14.11 Summary

This chapter has introduced the concepts of buckets and barriers as a means of providing synchronisation between processes that are executing on a single processor within a single JVM. It has been shown how an `AltingBarrier` can be used to manage highly dynamic situations and to provide a high-level control mechanism to manage complex interactions. A description of the implementation mechanism underlying `AltingBarrier` is to be found in (Welch, et al., 2007) and a different use of `AltingBarrier` using a syntactically different but conceptually identical formulation is to be found in (Ritson & Welch, 2007)

Index

To see Part II download
Using Concurrency and Parallelism Effectively – II